Portfolio Management with R

Enrico Schumann

15 October 2017

Contents

1 Introduction

1.1 About PMwR

This manual describes how to use the PMwR package. The aim of PMwR is to provide 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 currently under active development and changes frequently. This is mainly because the code has been written over many years and is in need of being groomed for general use. Consequently, this manual will change as frequently as the package.[1] I am grateful for comments and suggestions.

The latest version of the package is available from `http://enricoschumann.net/R/packages/PMwR/index.htm`. To install the package from within R, type

```
install.packages("PMwR",
                 type  = "source",
                 repos = c('http://enricoschumann.net/R',
                           getOption('repos')))
```

to download and install it. The package depends on several other packages, which are automatically obtained from the same repository and from CRAN. The source code is also pushed to a public repository at `https://github.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 via Rtools). If you have problems building the package for Windows, please contact me and I will provide you with a Windows version.

[1] The manual itself is written in Org mode. The complete tangled code is available from the website.

1.2 Principles

1.2.1 Small

The aim of PMWR is to provide a *small* set of tools. This comes at the price: interfaces may be more complicated. But with few functions, it is easier to remember a function name or to find it in the first place.

1.2.2 Flexible and general

PMWR aims to be open to different types of instruments, different timestamps, etc.

1.2.3 Functional

> *With properly designed functions, it is possible to ignore* how *a job is done; knowing* what *is done is sufficient.*
>
> *(K&R, chapter 1)*

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
- 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`.)

1.2.4 Matching by name

Whenever possible and intuitive, data should be matched by name, not by position. This is most natural with vectors that store scalar information about instruments, such as prices or multipliers. In such cases, data input such as prices) is preferred in the form of named vectors.

1.2.5 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` is internally computed through `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, `returns` can be computed very efficiently for a matrix.

1.3 Other packages

Several other packages originated from PMWR; initially, much of their code had been part of PMWR.

1.3.1 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/`

1.3.2 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`

1.3.3 tsdb

From the DESCRIPTION file:

A terribly-simple data base for time series. All series are saved as csv files. The package offers utilities for saving files in a standardised format, and for retrieving and joining data.

1.4 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")
```

2 Keeping track of transactions: journals

2.1 Overview

The ultimate basis of many financial computations are lists of transactions. And so many of the tools that the PMWR package provides take lists of transactions as input.

Conceptually, you can think of such lists as dataframes, but PMWR provides an `S3` class `journal` for handling them. A `journal` is a list of atomic vectors, to which a class attribute is attached. Such a list is created through the function `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 `amount` or `timestamp`, are called *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 `journal` at will; such dissections include removing the class attribute.

What is actually stored in a `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:

`amount` The notional amount that is transacted. `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 look at `amount`.

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

`price` Well, price.

`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 with only `NA` values in it, but `amount` 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 data frame to a journal, via a call to `as.journal`.

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

```
account => subaccount => subsubaccont => ... => instrument
```

This is useful and necessary when you have traded stock XY for different accounts, for instance, or as part of different strategies. Such a hierarchy may be completely captured in the instrument field, by concatenating account/instrument using a specific separator pattern such as `::`.[1] The result would be 'namespaced' instruments such as `Pension::Equities::AMZN`. Alternatively, part of the hierarchy may be stored in the `account` field.

2.2 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.

```
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.

```
journal()
```

[1] This notation is inspired by the syntax of `ledger` files. See `http://www.ledger-cli.org/` .

```
no transactions
```

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

```
position(jnl)
```

```
4
```

Only providing amounts is, admittedly, not overly useful. You can keep track of positions, true; but a journal implies chronological information, that is, flows. (As opposed to a ledger, which gives you positions, or stocks.)

When you make sure that the amounts are actually sorted in time, then you can at least track positions over time. (But nothing in the data structure that we created above could make sure that transactions really are sorted.)

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 may not have 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: perhaps checking 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, 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.

```
J <- journal(timestamp  = as.Date("2012-01-01") + 0:3,
             amount     = c(1, 2, -2, 5),
             instrument = c("EUR", "EUR", "CHF", "CHF"),
             comment    = c("initial balance", "",
                            "transfer", ""))
J
```

```
  instrument  timestamp  amount         comment
1        EUR 2012-01-01       1 initial balance
2        EUR 2012-01-02       2
3        CHF 2012-01-03      -2        transfer
4        CHF 2012-01-04       5

4 transactions
```

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`.

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

```
   timestamp  amount          comment
1  2012-01-01      1  initial balance
2  2012-01-02      2
[ .... ]

4 transactions
```

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

```
str(J)
```

```
'journal':       4 transactions
 $ instrument: chr [1:4] "EUR" "EUR" "CHF" "CHF"
 $ timestamp : Date[1:4], format: "2012-01-01" ...
 $ amount    : num [1:4] 1 2 -2 5
 $ price     : logi [1:4] NA NA NA NA
 $ comment   : chr [1:4] "initial balance" "" "transfer" ""
```

You may notice that the output is similar to that of a `data.frame` or `list`. That is because `J` is a `list` internally, with a class attribute. Essentially, it is little more than this:

```
list(timestamp  = as.Date("2012-01-01") + 0:3,
     amount     = c(1, 2, -2, 5),
     instrument = c("EUR", "EUR", "CHF", "CHF"),
     comment    = c("initial balance", "", "transfer", ""))
```

(But note that `journal` silently added a `price` field, even though we did not specify one.)

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).

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

```
            timestamp  amount
1  1501705632950756001      1
2  1501705632950756002      2
```

```
3  1501705632950756003       3

3 transactions
```

Journals can be combined with c.

```
J2 <- J
J2$fees <- rep(0.1,4)
c(J, J2)
```

```
   instrument   timestamp  amount          comment  fees
1         EUR  2012-01-01       1  initial balance    NA
2         EUR  2012-01-02       2                     NA
3         CHF  2012-01-03      -2         transfer    NA
4         CHF  2012-01-04       5                     NA
5         EUR  2012-01-01       1  initial balance   0.1
6         EUR  2012-01-02       2                    0.1
7         CHF  2012-01-03      -2         transfer   0.1
8         CHF  2012-01-04       5                    0.1

8 transactions
```

But we wanted the combined journal sorted by date.

```
sort(c(J, J2))
```

```
   instrument   timestamp  amount          comment  fees
1         EUR  2012-01-01       1  initial balance    NA
2         EUR  2012-01-01       1  initial balance   0.1
3         EUR  2012-01-02       2                     NA
4         EUR  2012-01-02       2                    0.1
5         CHF  2012-01-03      -2         transfer    NA
6         CHF  2012-01-03      -2         transfer   0.1
7         CHF  2012-01-04       5                     NA
8         CHF  2012-01-04       5                    0.1

8 transactions
```

We can also sort by some other field, such as amount.

```
sort(c(J, J2), by = "amount", decreasing = TRUE)
```

```
   instrument   timestamp  amount          comment  fees
1         CHF  2012-01-04       5                     NA
```

```
2           CHF   2012-01-04        5                     0.1
3           EUR   2012-01-02        2                      NA
4           EUR   2012-01-02        2                     0.1
5           EUR   2012-01-01        1  initial balance     NA
6           EUR   2012-01-01        1  initial balance    0.1
7           CHF   2012-01-03       -2         transfer     NA
8           CHF   2012-01-03       -2         transfer    0.1

8 transactions
```

2.3 Selecting transactions

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

```
subset(J, amount > 1)
```

```
   instrument   timestamp  amount  comment
1         EUR  2012-01-02       2
2         CHF  2012-01-04       5

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 `[[...]]`.[2]

```
J$amount
```

```
[1]  1  2 -2  5
```

You can also replace specific fields.

```
J[["amount"]] <- c(1 ,2, -2, 8)
J
```

[2]The 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 even though the commit was reversed two days later https://developer.r-project.org/blosxom.cgi/R-devel/NEWS/2016/03/11

```
   instrument   timestamp  amount          comment
1         EUR  2012-01-01       1  initial balance
2         EUR  2012-01-02       2
3         CHF  2012-01-03      -2         transfer
4         CHF  2012-01-04       8

4 transactions
```

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

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

```
   instrument   timestamp  amount   comment
1         EUR  2012-01-02       2
2         CHF  2012-01-03      -2  transfer

2 transactions
```

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

```
   instrument   timestamp  amount   comment
1         CHF  2012-01-03      -2  transfer

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["eur"]
```

```
   instrument   timestamp  amount          comment
1         EUR  2012-01-01       1  initial balance
2         EUR  2012-01-02       2

2 transactions
```

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

```
J["Transfer"]
```

```
   instrument   timestamp  amount   comment
1         CHF  2012-01-03      -2  transfer

1 transaction
```

```
J["Transfer", ignore.case = FALSE]
```

```
no transactions
```

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

```
J["Transfer", match.against = "instrument"]
```

```
no transactions
```

2.4 Computing balances

2.4.1 position

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

```
position(J)
```

```
    2012-01-04
CHF          6
EUR          3
```

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

```
position(J, when = as.Date("2012-01-03"))
```

```
    2012-01-03
CHF         -2
EUR          3
```

If you do not like such a tabular view, consider splitting the journal.

```
lapply(split(J, J$instrument),
       position, when = as.Date("2012-01-03"))
```

```
$CHF
    2012-01-03
CHF         -2

$EUR
    2012-01-03
EUR          3
```

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.

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

```
           CHF EUR
2012-01-01   0   1
2012-01-02   0   3
2012-01-03  -2   3
2012-01-04   3   3
```

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.) `endofmonth` prints the positions at the ends of all calendar months between the first and the last timestamp.

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

```
position(J, when = seq(from = as.Date("2011-12-30"),
                       to   = as.Date("2012-01-06"),
                       by   = "1 day"))
```

```
           CHF EUR
2011-12-30   0   0
2011-12-31   0   0
2012-01-01   0   1
2012-01-02   0   3
2012-01-03  -2   3
2012-01-04   6   3
2012-01-05   6   3
2012-01-06   6   3
```

By default, position will show you all positions, even if they are zero.

```
J <- c(J, journal(instrument = "EUR",
                  amount = -3,
                  timestamp = as.Date("2012-01-05")))
position(J)
```

```
    2012-01-05
CHF          6
EUR          0
```

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

```
position(J, drop.zero = TRUE)
```

```
    2012-01-05
CHF          6
```

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

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

```
    2012-01-05
CHF   6.00e+00
USD   2.78e-17
```

```
position(J, drop.zero = 1e-15)
```

```
    2012-01-05
CHF          6
```

TODO: position may also use the account field.

2.4.2 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.

```
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++];
}
```

2.5 Aggregating journal information

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`.

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 `jnl` and want to compute monthly turnover (two-way). If there is only one instrument or all instruments may be added without harm, you can use this expression:

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

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

```
tapply(jnl,
       INDEX = list(format(jnl$timestamp, "%Y-%m"),
                    jnl$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.

`aggregate.journal` 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 it is function, it should expect to 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/sells for every day.

```
jnl <- journal(timestamp = structure(c(15950, 15951, 15950, 15951, 15950,
                                       15950, 15951, 15951, 15951, 15951),
                                     class = "Date"),
```

```
                amount = c(-3, -4, -3, -1, 3, -2, 1, 3, 5, 3),
                price = c(104, 102, 102, 110, 106, 104, 104, 106, 108, 107),
                instrument = c("B", "B", "A", "A", "B", "B", "A", "B", "A", "A"))


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

```
   instrument   timestamp  amount     price
1           A  2013-09-02      -3  102.0000
2           B  2013-09-02      -5  104.0000
3           B  2013-09-02       3  106.0000
4           A  2013-09-03      -1  110.0000
5           B  2013-09-03      -4  102.0000
6           A  2013-09-03       9  107.2222
7           B  2013-09-03       3  106.0000

7 transactions
```

3 Computing profit and loss

In this chapter we will deal with computing profit and loss in amount of currency. If you are interested in computing returns, see Section Computing returns.

3.1 Simple cases

3.1.1 Total P/L

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 PMWR package provides a function `pl`, which for this case may be called as follows.

```
pl(price  = c(100, 101),
   amount = c(1, -1))
```

```
P/L total       1
average buy   100
average sell  101
cum. volume     2

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

Instead of a vectors `price` and `amount`, you could also have passed a journal to `pl`.

In principle, 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.

$$\sum x_i^{\text{sell}} p_i^{\text{sell}} - \sum x_i^{\text{buy}} p_i^{\text{buy}} \tag{3.1}$$

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

$$\begin{aligned} \text{P/L} &= -\sum_{x<0} x_i p_i - \sum_{x>0} x_i p_i & (3.2)\\ &= -\sum x_i p_i & (3.3) \end{aligned}$$

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.

```
amount <- rep(c(-100,100), 100)
price <- rep(100, length(amount))

library("rbenchmark")
benchmark(
    amount %*% price,
    sum(amount*price),
    crossprod(amount, price),
    t(amount*price) %*% rep(1, length(amount)),
    columns = c("test",    "elapsed", "relative"),
    order = "relative",
    replications = 20000)
```

`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.

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

```
   instrument  amount  price
1      Adidas      50    100
2      Adidas     -50    102
3 Commerzbank     500      8
4 Commerzbank    -500      7

4 transactions
```

We can now pass the journal directly to `pl`.

```
pl(jnl)
```

```
Adidas
  P/L total      100
  average buy    100
  average sell   102
  cum. volume    100

Commerzbank
  P/L total      -500
  average buy       8
  average sell      7
  cum. 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 price becomes less informative.

Financial instruments do not only differ 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).

```
jnl <- 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 |
")

jnl <- as.journal(jnl)
jnl
```

```
    instrument  amount    price
1  FGBL MAR 16       1   165.20
2  FGBL MAR 16      -1   165.37
3  FGBL JUN 16       1   164.12
```

```
4   FGBL  JUN  16       -1    164.13
5   FESX  JUN  16        5   2910.00
6   FESX  JUN  16       -5   2905.00

6 transactions
```

One point of the FGBL translates into 1000 euros; for the FESX it is 10 euros.

```
futures_pl <- pl(jnl,
                 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.

```
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.

Data can be extracted by standard methods. The vectors 'realised' and 'unrealised' will be `NA` unless `along.timestamp` is `TRUE`, also described in Section PL over time.

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

```
  pl   realised unrealised        buy       sell     volume
-250         NA         NA       2910       2905         10
```

```
unlist(lapply(futures_pl, `[[`, "volume"))
```

```
FESX JUN 16 FGBL JUN 16 FGBL MAR 16
         10           2           2
```

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:

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

```
FESX JUN 16 FGBL JUN 16 FGBL MAR 16
       -250          10         170
```

It is often more convenient to have the data presented as a table.

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

```
              pl     buy    sell volume
FESX JUN 16 -250 2910.00 2905.00     10
FGBL JUN 16   10  164.12  164.13      2
FGBL MAR 16  170  165.20  165.37      2
```

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

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

```
| instrument  |   pl |    buy |   sell | volume |
|-------------+------+--------+--------+--------|
| FESX JUN 16 | -250 |   2910 |   2905 |     10 |
| FGBL JUN 16 |   10 | 164.12 | 164.13 |      2 |
| FGBL MAR 16 |  170 |  165.2 | 165.37 |      2 |
```

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

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

```
P/L total      NA
average buy   100
average sell   NA
cum. volume     1

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

Warning message:
```

```
In pl.default(amount = 1, price = 100) :
  'sum(amount)' is not zero: specify 'vprice' to compute p/l
```

Of course, there be no P/L number. But the warning message that is thrown already 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`.

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

```
P/L total        5
average buy    100
average sell   105
cum. volume      1

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

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.

```
open_position <- c(`FESX JUN 16` = -20, `FGBL JUN 16` = 10)
prices        <- c(`FESX JUN 16` = 2912, `FGBL JUN 16` = 164.23)
```

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.

```
jnl
```

```
   instrument  amount   price
1  FGBL MAR 16       1  165.20
2  FGBL MAR 16      -1  165.37
3  FGBL JUN 16       1  164.12
4  FGBL JUN 16      -1  164.13
```

```
5  FESX JUN 16      5  2910.00
6  FESX JUN 16     -5  2905.00

6 transactions
```

```
pl(jnl,
   initial.position = open_position,
   initial.price = 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.
```

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. But it is often 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).

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

```
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(jnl, along.timestamp = TRUE)
```

```
P/L total        0 -40  60
__ realised      0   0  60
__ unrealised    0 -40   0
average buy     70
average sell   100
volume           1   2   4

'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 implicitly uses integers `1`, `2`, …, `length(amount)`. 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.

```
str(pl(jnl, along.timestamp = TRUE))
```

```
List of 1
 $ :List of 7
  ..$ timestamp : logi [1:3] NA NA NA
  ..$ 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
```

In the previous section, we used `vprice` to value a final open position. It turns out we can also use it to value a position over time.

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 also instruments with variable multipliers, e.g. Australian government bond futures. An easy to handle this is by computing effective position sizes; but it may take some thinking to come up with a reusable scheme (e.g., looking up multipliers in a table).
- 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 often will often 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).

4 Computing returns

4.1 Simple returns

The function `returns` computes returns from prices. The function computes what are sometimes called simple returns:[1] let P_t be the price at point in time t, then

$$r_t \equiv R_t - 1 = \frac{P_t}{P_{t-1}} - 1 = \frac{P_t - P_{t-1}}{P_{t-1}}\,. \tag{4.1}$$

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:

```
simple_returns <- function(x)
    x[-1L]/x[-length(x)] - 1
```

(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. PMWR comes with two small datasets, DAX and REXP. Both are data-frames of one column; the rownames are the dates. Given a vector of prices – here, the closing values of the DAX, the German stock-market index, for the first five business days of 2014, – the function computes returns.

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

```
[1] 9400 9435 9428 9506 9498
```

```
simple_returns(P)
```

```
[1]  0.003735 -0.000758  0.008294 -0.000879
```

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

[1]The function never computes logarithmic returns.

```
returns(P)
```

```
[1]  0.003735 -0.000758  0.008294 -0.000879
```

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]

```
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
```

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.

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

```
  price   returns
1  9400        NA
2  9435  0.003735
3  9428 -0.000758
4  9506  0.008294
5  9498 -0.000879
```

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.)

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

```
[1] 9400 9435 9428 9506 9498
```

`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`. This function is the actual workhorse that performs the raw returns calculation.

[2] See section Vectorisation.

Besides having methods for numeric vectors and dataframes, `returns` also understands `zoo` objects.

So let us create two `zoo` series, `DAX` and `REXP`. These two variables now mask the original dataframes in PMWR. To get the latter back, either remove the variables or say `data(DAX)`.

```
DAX <- zoo(DAX[[1]], as.Date(row.names(DAX)))
REXP <- 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" ...
```

```
str(REXP)
```

```
'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)
```

```
2014-01-02 2014-01-03 2014-01-06 2014-01-07 2014-01-08
        NA   0.003735  -0.000758   0.008294  -0.000879
```

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

```
drax <- cbind(DAX, REXP)
returns(head(drax, 5))
```

```
                 DAX      REXP
2014-01-03  0.003735  0.000611
2014-01-06 -0.000758  0.001704
2014-01-07  0.008294  0.000621
2014-01-08 -0.000879 -0.000131
```

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.)

[3]In case you did not know: `drax` is not only the name of a dataset in this book, but also the name of superhero and of the villain of a James Bond novel. The latter is actually German, which makes it obvious to choose his name for representing German indices.

4.2 Holding-period returns

When a timestamp is available, `returns` can compute returns for specific calendar periods. As an example, we look at the daily DAX levels in 2014 and 2015.

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

```
       Jan  Feb   Mar   Apr   May   Jun   Jul   Aug   Sep   Oct  Nov   Dec  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
```

If, for some reason, you prefer not use `zoo`, you can also pass the timestamp explicitly to `returns`.

```
returns(coredata(DAX), t = index(DAX), period = "month")
```

```
       Jan  Feb   Mar   Apr   May   Jun   Jul   Aug   Sep   Oct  Nov   Dec  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
```

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. computing means, extremes or similar quantities.

```
range(returns(DAX, period = "month"))
```

```
[1] -0.0928  0.1232
```

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. Suppose we combine the prices of the DAX and of the REXP. The function's assumption is that now it would be more convenient to print the returns aligned by date in a table.

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

```
             DAX   REXP
2014-01-31  -1.0    1.8
2014-02-28   4.1    0.4
2014-03-31  -1.4    0.1
2014-04-30   0.5    0.3
2014-05-30   3.5    0.9
```

```
2014-06-30   -1.1    0.4
2014-07-31   -4.3    0.4
2014-08-29    0.7    1.0
2014-09-30    0.0   -0.1
2014-10-31   -1.6    0.1
2014-11-28    7.0    0.4
2014-12-30   -1.8    1.0
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.

```
lapply(list(DAX = dax, REXP = rex),
       returns, period = "month")
```

```
$DAX
      Jan  Feb   Mar   Apr   May   Jun   Jul   Aug   Sep   Oct  Nov   Dec  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

$REXP
     Jan  Feb  Mar   Apr   May   Jun  Jul  Aug   Sep  Oct  Nov   Dec  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
```

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

```
print(returns(DAX, period = "month"),
      digits = 2, year.rows = FALSE, plus = TRUE,
      month.names = 1:12)
```

```
      2014     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
11    +7.01   +4.90
12    -1.76   -5.62
YTD   +4.31   +9.56
```

There are methods `toLatex` and `toHTML` for monthly returns.

Sweave In Sweave documents, you need to use `results = tex` and `echo = false` in the chunk options:

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

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

`returns` accepts other values for `period`. For yearly returns, use `period "year"`.

```
returns(DAX, period = "year")
```

```
2014  2015
 4.3   9.6
```

```
returns(drax, period = "year")
```

```
     DAX  REXP
2014 4.3   7.1
2015 9.6   0.5
```

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

```
returns(DAX, period = "ann")
```

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

Now let us try a shorter period.

```
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 can see the monthly return for January 2014 in the tables above.)

To force annualising, add a `!`. The exclamation mark serves as a mnenomic that it is now imperative to annualise.

```
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 month-to-date (`mtd`), year-to-date (`ytd`) or inception-to-date (`itd`). The help page of `returns` lists all options.

4.3 Returns when weights are fixed

Sometimes we may need to compute returns 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 $[0.1, 0.5, 0.4]'$ to three funds would have done, assuming that a portfolio is rebalanced once a month.

If more detail is necessary, then `btest` can be used; see Chapter Backtesting. But the simple case can be done with `returns` already. Here is an example.

```
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  2.0  3.5
[2,]  102  2.2  3.0
[3,]  104  2.4  3.1
[4,]  104  2.3  3.2
[5,]  104  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.)

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

```
[1] -0.00514  0.06376 -0.01282  0.03146

attr(,"holdings")
      [,1]  [,2]  [,3]
[1,] 0.001 0.250 0.114
[2,] 0.001 0.250 0.114
[3,] 0.001 0.250 0.114
[4,] 0.001 0.227 0.131
[5,] 0.001 0.227 0.131

attr(,"contributions")
         [,1]    [,2]    [,3]
[1,] 0.000000  0.0000  0.0000
[2,] 0.002000  0.0500 -0.0571
[3,] 0.002010  0.0503  0.0115
[4,] 0.000000 -0.0236  0.0108
[5,] 0.000481  0.0435 -0.0125
```

In fact, rebalancing at the prices in 1 is always implied.

The result is the return series plus two additional pieces of information, stored in attributes.

holdings A matrix with the same dimensions as the price matrix we used as input. It provides the hypothetical holdings that were used to compute the returns. Note that these holdings only change at timestamps 1 and 4 in the example.

contributions Another matrix; it provides the return contributions of the single assets (in columns) in each period (in rows).

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.

$$r_P(t) = \sum_{i=1}^{N} r(t,i)w(t,i)\,. \tag{4.2}$$

When the weights sum to unity, we may also write

$$1 + r_P(t) = \sum_{i=1}^{N} \big(1 + r(t,i)\big)\, w(t,i) \tag{4.3}$$

or, defining $1 + r \equiv R$,

$$R_P(t) = \sum_{i=1}^{N} R(t,i)w(t,i)\,. \tag{4.4}$$

The total return contribution of segment i over time equals

$$\sum_{t=1}^{T-1} \left(R(t,i)w(t,i) \prod_{s=t+1}^{T} R_P(s) - 1 \right) + \underbrace{r(T,i)\, w(T,i)}_{\text{final period}}\,. \tag{4.5}$$

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

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

```
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         1   0.0025  0.00000  0.00250
2         2   0.0100 -0.00600  0.00400
3         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

The function `unit_prices` helps to compute 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.)

```
NAV <- data.frame(timestamp = seq(as.Date("2017-1-1"),
                                  as.Date("2017-1-10"),
                                  by = "1 day"),
                  NAV = c(0,101:104,205:209))

cf <- data.frame(timestamp = c(as.Date("2017-1-1"),
                               as.Date("2017-1-5")),
                 cashflow = c(100, 100))

unit_prices(NAV, cf)
```

```
    timestamp NAV   price  shares cashflow new_shares total_shares NAV_after_cf
1  2017-01-01   0 100.000 0.00000      100   1.000000      1.00000
100
2  2017-01-02 101 101.000 1.00000        0   0.000000      1.00000
101
3  2017-01-03 102 102.000 1.00000        0   0.000000      1.00000
102
4  2017-01-04 103 103.000 1.00000        0   0.000000      1.00000
103
5  2017-01-05 104 104.000 1.00000      100   0.961538      1.96154
204
6  2017-01-06 205 104.510 1.96154        0   0.000000      1.96154
205
7  2017-01-07 206 105.020 1.96154        0   0.000000      1.96154
206
8  2017-01-08 207 105.529 1.96154        0   0.000000      1.96154
207
9  2017-01-09 208 106.039 1.96154        0   0.000000      1.96154
208
```

```
10 2017-01-10 209 106.549 1.96154         0    0.000000       1.96154
209
```

5 Backtesting

This chapter explains how to test trading strategies with the `btest` function.

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 `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 `btest` is a loop that runs through the data. Loops are slow in R when compared with compiled languages, 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 `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 `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 `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: `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:

```
 open      high      low      close
+-+-+     +-+-+     +-+-+     +-+-+
| | |     | | |     | | |     | | |
| | |     | | |     | | |     | | |
| | |     | | |     | | |     | | |
| | |     | | |     | | |     | | |
| | |     | | |     | | |     | | |
+-+-+     +-+-+     +-+-+     +-+-+
```

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 `prices = list(p)`.

The `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:

```
t    time       open   high   low    close
1    HH:MM:SS                                  <--\
2    HH:MM:SS                                  <-- - use information
```

```
3    HH:MM:SS  _________________________  <--/
4    HH:MM:SS    X                        <- trade here
5    HH:MM:SS
```

We could also trade at the close:

```
t    time      open  high  low   close
1    HH:MM:SS                             <-- \
2    HH:MM:SS                             <-- - use information
3    HH:MM:SS  _________________________  <-- /
4    HH:MM:SS                       X     <-- trade here
5    HH:MM:SS
```

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 following 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

```
Close()
```

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

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

```
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

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

and saying

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

will get you the last ten closing prices.

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

- 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` or `lastofmonth` (in this case, `timestamp` must inherit from `Date` or be coercible to `Date`).

`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 that 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

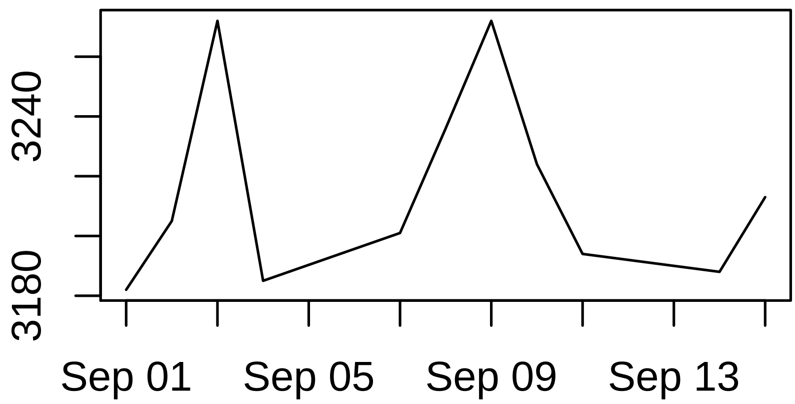
It is best to describe the `btest` function through a number of simple examples.

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.

```
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)
```

```
    timestamp prices
1  2015-09-01   3182
2  2015-09-02   3205
3  2015-09-03   3272
4  2015-09-04   3185
5  2015-09-07   3201
6  2015-09-08   3236
7  2015-09-09   3272
8  2015-09-10   3224
9  2015-09-11   3194
10 2015-09-14   3188
11 2015-09-15   3213
```



The `signal` function is very simple indeed.

```
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`.

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

```
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.

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

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

```
   price suggest position wealth  cash
1   3182       0        0      0     0
2   3205       1        1      0 -3205
3   3272       1        1     67 -3205
4   3185       1        1    -20 -3205
5   3201       1        1     -4 -3205
6   3236       1        1     31 -3205
7   3272       1        1     67 -3205
8   3224       1        1     19 -3205
9   3194       1        1    -11 -3205
10  3188       1        1    -17 -3205
11  3213       1        1      8 -3205
```

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.

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

```
   price  suggest  position  wealth   cash
1   3182        1         1       0  -3182
2   3205        1         1      23  -3182
3   3272        1         1      90  -3182
4   3185        1         1       3  -3182
5   3201        1         1      19  -3182
6   3236        1         1      54  -3182
7   3272        1         1      90  -3182
8   3224        1         1      42  -3182
9   3194        1         1      12  -3182
10  3188        1         1       6  -3182
11  3213        1         1      31  -3182
```

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 it the price is above

1. The signal function could look like this.

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

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

```
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.[1]

We call btest and check the results.

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

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

```
  price suggest position wealth  cash
1  3182       0        0      0     0
2  3205       1        1      0 -3205
3  3272       0        0     67    67
4  3185       0        0     67    67
5  3201       1        1     67 -3134
6  3236       0        0    102   102
```

[1]Remember 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?*

```
7    3272        0        0     102    102
8    3224        0        0     102    102
9    3194        0        0     102    102
10   3188        1        1     102  -3086
11   3213        1        1     127  -3086
```

(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.

```
solution <- btest(prices = prices, signal = signal,
                  initial.position = 1)
```

Then the results would have looked as follows.

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

```
   price suggest position wealth cash
1   3182       1        1   3182    0
2   3205       1        1   3205    0
3   3272       0        0   3272 3272
4   3185       0        0   3272 3272
5   3201       1        1   3272   71
6   3236       0        0   3307 3307
7   3272       0        0   3307 3307
8   3224       0        0   3307 3307
9   3194       0        0   3307 3307
10  3188       1        1   3307  119
11  3213       1        1   3332  119
```

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:

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

```
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`.

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

solution <- btest(prices = prices,
                  signal = signal,
                  threshold = 3190)

trade_details(solution, prices)
```

```
   price suggest position wealth  cash
1   3182       0        0      0     0
2   3205       1        1      0 -3205
3   3272       0        0     67    67
4   3185       0        0     67    67
5   3201       1        1     67 -3134
6   3236       0        0    102   102
7   3272       0        0    102   102
8   3224       0        0    102   102
9   3194       0        0    102   102
10  3188       0        0    102   102
11  3213       1        1    102 -3111
```

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)
```

```
   price  suggest  position  wealth   cash
1   3182        0        NA      NA      0
2   3205        0         0       0      0
3   3272        0         0       0      0
4   3185        0         0       0      0
5   3201        1         1       0  -3201
6   3236        0         0      35     35
7   3272        0         0      35     35
8   3224        0         0      35     35
9   3194        1         1      35  -3159
10  3188        1         1      29  -3159
11  3213        1         1      54  -3159
```

If we want to trade a different size, we have `signal` return the desired value.

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

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

```
   price  suggest  position  wealth   cash
1   3182        0         0       0      0
2   3205        2         2       0  -6410
3   3272        0         0     134    134
4   3185        0         0     134    134
5   3201        2         2     134  -6268
6   3236        0         0     204    204
7   3272        0         0     204    204
8   3224        0         0     204    204
9   3194        0         0     204    204
10  3188        2         2     204  -6172
11  3213        2         2     254  -6172
```

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'.

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

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

```
              prices)
```

```
   price  suggest  position  wealth   cash
1   3182        0         0       0      0
2   3205        0         0       0      0
3   3272        0         0       0      0
4   3185        1         1       0  -3185
5   3201        0         0      16     16
6   3236        0         0      16     16
7   3272        0         0      16     16
8   3224        0         0      16     16
9   3194        0         0      16     16
10  3188        0         0      16     16
11  3213        0         0      16     16
```

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.

```
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.

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

```
   prices  sp  asset.1  wealth   cash
1    3182  0         0       0      0
2    3205  0         0       0      0
3    3272  0         0       0      0
4    3185  1         1       0  -3185
5    3201  1         1      16  -3185
6    3236  1         1      51  -3185
7    3272  1         1      87  -3185
8    3224  1         1      39  -3185
9    3194  1         1       9  -3185
10   3188  1         1       3  -3185
11   3213  1         1      28  -3185
```

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.)

```
signal <- function()
    0.05

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

```
   prices      sp asset.1 wealth  cash
1    3182 0.00000 0.00000    100 100.0
2    3205 0.00157 0.00157    100  95.0
3    3272 0.00156 0.00156    100  95.0
4    3185 0.00153 0.00153    100  95.1
5    3201 0.00157 0.00157    100  95.0
6    3236 0.00156 0.00157    100  95.0
7    3272 0.00155 0.00155    100  95.0
8    3224 0.00153 0.00153    100  95.1
9    3194 0.00155 0.00155    100  95.0
10   3188 0.00157 0.00157    100  95.0
11   3213 0.00157 0.00157    100  95.0
```

Note that until now we – potentially – rebalanced in every period. If you do not want that, we need to specify `do.rebalance`.

```
do.rebalance <- function() {
    if (sum(abs(
        SuggestedPortfolio(0) - SuggestedPortfolio())) > 2e-5)
        TRUE
    else
        FALSE
}

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

trade_details(solution, prices)
```

```
   price suggest position wealth   cash
1   3182  0.0000   0.0000    100 100.00
```

```
2    3205  0.0000   0.0000     100  100.00
3    3272  0.0000   0.0000     100  100.00
4    3185  0.0306   0.0306     100    2.66
5    3201  0.0000   0.0000     100  100.49
6    3236  0.0000   0.0000     100  100.49
7    3272  0.0000   0.0000     100  100.49
8    3224  0.0000   0.0000     100  100.49
9    3194  0.0000   0.0000     100  100.49
10   3188  0.0000   0.0000     100  100.49
11   3213  0.0000   0.0000     100  100.49
```

`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.

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

trade_details(solution, prices)
```

```
   prices      sp asset.1 wealth  cash
1    3182 0.00000 0.00000    100 100.0
2    3205 0.00157 0.00157    100  95.0
3    3272 0.00156 0.00157    100  95.0
4    3185 0.00153 0.00153    100  95.1
5    3201 0.00157 0.00157    100  95.0
6    3236 0.00156 0.00157    100  95.0
7    3272 0.00155 0.00155    100  95.0
8    3224 0.00153 0.00155    100  95.0
9    3194 0.00155 0.00155    100  95.0
10   3188 0.00157 0.00155    100  95.0
11   3213 0.00157 0.00157    100  95.0
```

Passing environments

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

```
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 {
        0
    }
}

solution <- btest(prices = prices,
                  signal = signal,
                  threshold = 3200,
                  notepad = notepad)

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

```
   price suggest position wealth  cash entry
1   3182       0        0      0     0     0
2   3205       1        1      0 -3205  3205
3   3272       0        0     67    67  3205
4   3185       0        0     67    67  3205
5   3201       1        1     67 -3134  3201
6   3236       0        0    102   102  3201
7   3272       0        0    102   102  3201
8   3224       0        0    102   102  3201
9   3194       0        0    102   102  3201
10  3188       1        1    102 -3086  3188
11  3213       1        1    127 -3086  3188
```

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)
```

```
   price suggest position wealth  cash entry
1   3182       0        0      0     0    NA
2   3205       1        1      0 -3205  3205
3   3272       0        0     67    67  3205
4   3185       0        0     67    67  3205
5   3201       1        1     67 -3134  3201
6   3236       0        0    102   102  3201
7   3272       0        0    102   102  3201
8   3224       0        0    102   102  3201
9   3194       0        0    102   102  3201
10  3188       1        1    102 -3086  3188
11  3213       1        1    127 -3086  3188
```

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.

5.5.1 A simple example

```
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)
```

```
   price.A price.B suggest.A suggest.B position.A position.B wealth cash
1      100     100         0         0         NA         NA     NA
0
2       98      99         0         0          0          0      0
0
3       98     100         0         1          0          1      0 -100
4       97     102         0         1          0          1      2 -100
5       96     101         0         1          0          1      1 -100
6       98     100         0         1          0          1      0 -100
7       97      96         0         1          0          1     -4 -100
8       98      97         2         0          2          0     -3 -199
9       99      95         2         0          2          0     -1 -199
10     101      82         2         0          2          0      3 -199
```

```
journal(solution)
```

```
   instrument  timestamp  amount  price
1           B          3       1    100
2           A          8       2     98
3           B          8      -1     97

3 transactions
```

5.6 Common tasks

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

5.6.1 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.)

5.6.2 Delaying signals

Add a random variable to to `do.rebalance`:

```
if (runif(1) > prob_of_delay)
    TRUE else FALSE
```

If `TRUE`, rebalancing will take place.

5.6.3 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 dates: 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 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 `firstofmonth` and `lastofmonth`:

```
btest(prices = prices,
      signal = signal,
      do.signal = "firstofmonth")
```

5.6.4 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

```
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.

5.6.5 Selecting parameters: calling `btest` recursively

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

```
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:

```
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 θ (or NULL). When the function calls signal, θ is not defined and signal will call btest with a specified θ.

Let us look at an actual example.

```
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
}

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)
```

6 Rebalancing a portfolio

The function `rebalance` computes the transactions necessary for moving from one portfolio to another.

6.1 Usage with unnamed vectors

The `current` portfolio is given in currency units; the `target` portfolio is given 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.

```
current <- c(0,0,100,100)
prices  <- c(1,1,1,1)
target  <- c(0.25, 0.25, 0.25, 0.25)
rebalance(current, target, prices, match.names = FALSE)
```

```
  price current value    %      target value    %      order
1     1       0     0  0.0          50    50 25.0         50
2     1       0     0  0.0          50    50 25.0         50
3     1     100   100 50.0          50    50 25.0        -50
4     1     100   100 50.0          50    50 25.0        -50

Notional: 200.  Amount invested: 200.  Total (2-way) turnover: 200.
```

The current portfolio may also be empty, in which case `current` can be set to `0`. Then, of course, we need to specify a `notional` for the target portfolio.

```
current <- 0
rebalance(current, target, prices,
          match.names = FALSE, notional = 100)
```

```
  price current value   %      target value    %      order
1     1       0     0 0.0          25    25 25.0         25
2     1       0     0 0.0          25    25 25.0         25
3     1       0     0 0.0          25    25 25.0         25
4     1       0     0 0.0          25    25 25.0         25
```

```
Notional: 100.  Amount invested: 100.  Total (2-way) turnover: 100.
```

We may also specify the target portfolio as a single number.

```
current <- c(5, 5, 100, 100)

target <- 0    ## liquidate the portfolio
rebalance(current, target, prices, match.names = FALSE)
```

```
  price current value    %       target value   %       order
1     1       5     5  2.4            0     0 0.0          -5
2     1       5     5  2.4            0     0 0.0          -5
3     1     100   100 47.6            0     0 0.0        -100
4     1     100   100 47.6            0     0 0.0        -100

Notional: 210.  Amount invested: 0.  Total (2-way) turnover: 210.
```

```
## every assets gets a weight of 20%
target <- 0.2
rebalance(current, target, prices, match.names = FALSE, notional = 100)
```

```
  price current value     %       target value    %       order
1     1       5     5   5.0           20    20 20.0          15
2     1       5     5   5.0           20    20 20.0          15
3     1     100   100 100.0           20    20 20.0         -80
4     1     100   100 100.0           20    20 20.0         -80

Notional: 100.  Amount invested: 80.  Total (2-way) turnover: 190.
```

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)
```

```
  price current value     %       target value    %       order
b     1      10    10 100.0            0     0  0.0         -10
d     1       0     0   0.0            5     5 50.0           5

Notional: 10.  Amount invested: 5.  Total (2-way) turnover: 15.
```

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

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

```
  price current value     %       target value    %       order
a     1       0     0   0.0            0     0  0.0           0
b     1      10    10 100.0            0     0  0.0         -10
d     1       0     0   0.0            5     5 50.0           5

Notional: 10.  Amount invested: 5.  Total (2-way) turnover: 15.
```

6.3 Optimisation

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.

```
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)
```

```
   price lot.size
1    178        5
2     37        5
3     62        5
4     93        1
5     81        5
6    111        5
7    146        5
8    154        5
9    187        1
10   138        1
```

Now suppose we have only a limited budget available.

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

```
   price  current  value    %       target  value     %        order
1    178        0      0  0.0            4    712   7.1            4
2     37        0      0  0.0           40   1480  14.8           40
3     62        0      0  0.0           20   1240  12.4           20
4     93        0      0  0.0           16   1488  14.9           16
5     81        0      0  0.0           13   1053  10.5           13
6    111        0      0  0.0            6    666   6.7            6
7    146        0      0  0.0            4    584   5.8            4
8    154        0      0  0.0            6    924   9.2            6
9    187        0      0  0.0            5    935   9.3            5
10   138        0      0  0.0            7    966   9.7            7

Notional: 10000.  Amount invested: 10048.  Total (2-way) turnover: 10048.
```

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)), ]
```

```
   TA rounded
1   5       5
2  40      40
3  20      20
4  16      16
5  15      15
6   5       5
7   5       5
8   5       5
9   5       5
10  7       7
```

The difference.

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

```
[1] 0
```

6.4 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:

```
w <- c(basket_1 = 0.3,
       basket_2 = 0.5,
       basket_3 = 0.2)
```

We also know what the first two baskets represent.

```
b1 <- c(a = 0.5, b = 0.2, c = 0.3)
b2 <- c(d = 0.1, e = 0.2, a = 0.7)
```

Now we can call `replace_weight`.

```
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.

```
replace_weight(c("basket 1" = 0.3,
                 "basket 2" = 0.7),
               "basket 1" = b1,
               "basket 2" = b2)
```

7 Summarising portfolio time-series

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

To handle such series, PMWR uses 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; while 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 `NAVseries`

The PMWR package provide a data frame `DAX`. DAX stands for *Deutscher Aktienindex* (German Equity Index), and the data frame contains closing prices of the index, with the timestamps stored as rownames.

```
str(DAX)
head(DAX)
```

```
'data.frame':   505 obs. of  1 variable:
 $ DAX: num  9400 9435 9428 9506 9498 ...
               DAX
2014-01-02 9400.04
2014-01-03 9435.15
2014-01-06 9428.00
2014-01-07 9506.20
2014-01-08 9497.84
2014-01-09 9421.61
```

We first transform the data frame into an `NAVseries` by calling the function of the same name.

```
dax <- NAVseries(DAX[[1]], as.Date(row.names(DAX)), title = "DAX")
dax
```

```
DAX
02 Jan 2014 ==> 30 Dec 2015   (505 data points, 0 NAs)
    9400.04           10743
```

7.2 Methods

Most useful is probably the summary method.

```
summary(dax)
```

```
---------------------------------------------------------
DAX
02 Jan 2014 ==> 30 Dec 2015   (505 data points, 0 NAs)
    9400.04           10743
---------------------------------------------------------
High                  12374.73  (10 Apr 2015)
Low                    8571.95  (15 Oct 2014)
---------------------------------------------------------
Return (%)                 6.9  (annualised)
---------------------------------------------------------
Max. drawdown (%)         23.8
_ peak                12374.73  (10 Apr 2015)
_ trough               9427.64  (24 Sep 2015)
_ underwater now (%)      13.2
---------------------------------------------------------
Volatility (%)            18.0  (annualised)
_ upside                  14.4
_ downside                10.4
---------------------------------------------------------

Monthly returns

      Jan Feb  Mar  Apr  May  Jun  Jul  Aug  Sep  Oct Nov  Dec 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
```

There are a few other methods, e.g. to coercion to `zoo`. There is als a generic `as.NAVseries` for coercion to an `NAVseries`.

There is also a convenience method for `btest` objects (see Chapter Backtesting), which extract the equity series from backtests.

For summaries of NAV series, a method for `toLatex` can be used to fill LaTeX-templates. The package comes with a vignette that provides examples.

8 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 at pre-defined, usually equally-spaced points in time, but rather split the trading history the history into separate trades.

8.1 Exposure

We have the following trades and times.

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

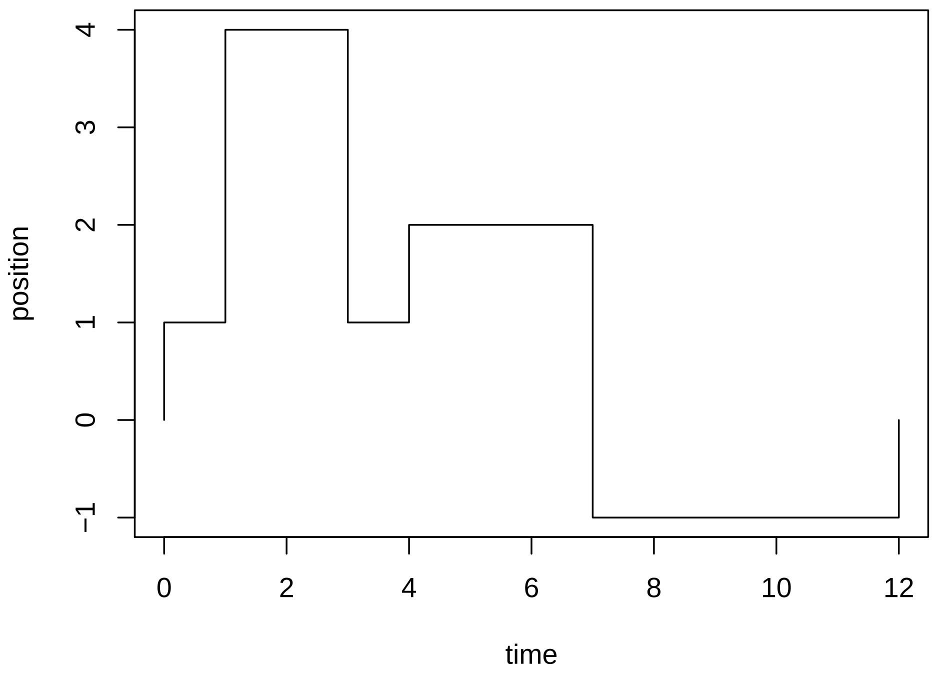
The holding period (duration) of these trades can be computed so:

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

```
  position from to duration
1        1    0  1        1
2        4    1  3        2
3        1    3  4        1
4        2    4  7        3
5       -1    7 12        5
```

We can plot the exposure.

```
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 `twExposure` (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
```

8.2 Splitting and rescaling

We have the following trades.

```
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`.

```
split_trades(amount, price, 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: one contract sold closes the first trade; the other contract opens the second trade.

That is useful in its own right: there are accounting systems around that cannot handle a trade switches a trade 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 that change the position sign splitted.

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

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

$price
[1] 100  99  99 101

$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.

```
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)
```

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

$price
[1] 100  99  99 100

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

Scaling the trades.

```
scale_to_unity(n)
```

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

Closing the trade at once.

```
close_on_first(n)
```

```
[1] -1 -1 -1  3  0  0
```

9 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 is becomes easier to compare them. It is a generic function, and `PMwR` provides methods for numeric vectors/matrices, and for `zoo` and `NAVseries` objects.

9.1 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.

```
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.

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

```
[1] 1.000000 1.003735 1.002974 1.011294 1.010404
[1] 1.000000 1.003735 1.002974 1.011294 1.010404
```

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

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

```
scale1(dax, level = 100)
```

```
[1] 100.0000 100.3735 100.2974 101.1294 101.0404
```

[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).

If we give a matrix to `scale1`, the function scales each column separately.

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

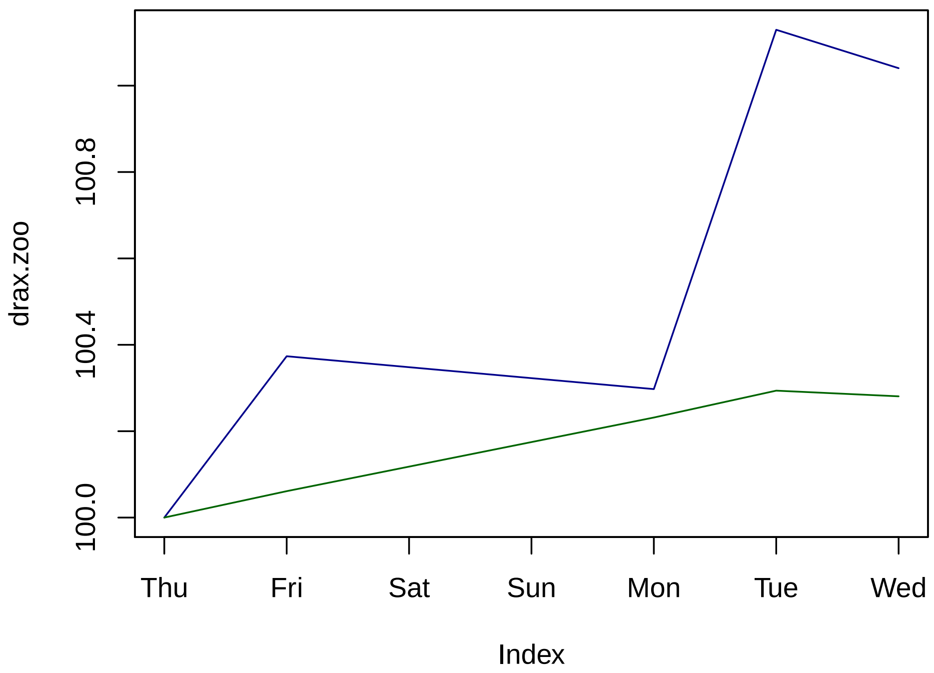
```
            dax      rexp
[1,] 100.0000 100.0000
[2,] 100.3735 100.0611
[3,] 100.2974 100.2316
[4,] 101.1294 100.2939
[5,] 101.0404 100.2807
```

`scale1` is a generic function; it works, for instance, with `zoo` objects.

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

```
                dax      rexp
2014-01-02 100.0000 100.0000
2014-01-03 100.3735 100.0611
2014-01-06 100.2974 100.2316
2014-01-07 101.1294 100.2939
2014-01-08 101.0404 100.2807
```

```
plot(drax.zoo, plot.type = "single", col = c("darkblue", "darkgreen"))
```



The argument when defines the origin.

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

```
            dax       rexp
[1,]  99.70344  99.76890
[2,] 100.07584  99.82987
[3,] 100.00000 100.00000
[4,] 100.82944 100.06208
[5,] 100.74077 100.04899
```

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  98.88326  99.70701
2014-01-03  99.25259  99.76794
2014-01-06  99.17738  99.93796
```

```
2014-01-07 100.00000 100.00000
2014-01-08  99.91206  99.98692
```

`when` also understands the keyword `first.complete`, which is actually the default. That is useful when series have different lengths.

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

```
         dax     rexp
[1,]      NA 440.5252
[2,]      NA 440.7944
[3,] 9428.00 441.5456
[4,] 9506.20 441.8197
[5,] 9497.84 441.7619
```

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

```
          dax      rexp
[1,]       NA  99.76890
[2,]       NA  99.82987
[3,] 100.0000 100.00000
[4,] 100.8294 100.06208
[5,] 100.7408 100.04899
```

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

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

```
                 dax     rexp
2014-01-02 1.0000000 1.000000
2014-01-03 1.0011441 0.999910
2014-01-06 0.9977916 1.000913
2014-01-07 1.0034825 1.000833
2014-01-08 1.0000000 1.000000
```

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.

```
apply(returns(scale1(drax.zoo, scale = 0.02)), 2, sd)
```

```
 dax rexp
0.02 0.02
```

This may create fairer comparisons, for instance, between fund prices that exhibit very different volatilities.

```
scale1(drax.zoo, scale = 0.02)
```

```
                dax     rexp
2014-01-02 1.000000 1.000000
2014-01-03 1.017123 1.016175
2014-01-06 1.013590 1.062012
2014-01-07 1.052132 1.079462
2014-01-08 1.047890 1.075724
```

It should be stressed that `centre` and `scale` treat *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.

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

```
2015-01-01 2015-01-02 2015-01-03 2015-01-04 2015-01-05 2015-01-06
       100        100        100        100        100        100
2015-12-27 2015-12-28 2015-12-29 2015-12-30 2015-12-31 2016-01-01
       100        100        100        100        100        100
```

`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).

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

```
2015-01-01 2015-01-02 2015-01-03 2015-01-04 2015-01-05 2015-01-06
  1.000000   1.000054   1.000109   1.000163   1.000217   1.000271
2015-12-27 2015-12-28 2015-12-29 2015-12-30 2015-12-31 2016-01-01
  1.019723   1.019779   1.019834   1.019889   1.019945   1.020000
```

9.2 Scaling a series

The previous section provided examples of scaling series. In this section, we are going to see how `scale1` does its computations.

First, a series P passed to `scale1` is transformed into returns, R. The `scale` argument allows you to set a desired **volatility**, defined as the standard deviation, for the returns. 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.

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

$$\frac{P_1}{P_0}\frac{P_2}{P_1}\cdots\frac{P_T}{P_{T-1}} = \frac{P_T}{P_0} = (1+r_1)(1+r_2)(1+r_3)\cdots = \prod_{t=1}^{T} 1+r_t \tag{9.1}$$

There clearly is an infinity of possible adjustments that would do the trick. We might, for instance, change 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. Define r_* as the desired total return, we need to solve the following equation for z.

$$(1+r_1+z)(1+r_2+z)(1+r_3+z)\cdots = 1+r_* \tag{9.2}$$

Alternatively, we may use logs.

$$\sum_i \log(1+r_i+z) = \log(1+r_*) \tag{9.3}$$

This is a classical application for root-finding (see chapter 11 of (Gilli, Maringer, and Schumann, 2011)), for which we use `uniroot`.

```
P1 <- cumprod(1 + c(0, rnorm(20, sd = 0.02)))
P1_scaled <- scale1(P1, centre = TRUE)

sd(returns(P1))
sd(returns(P1_scaled))

tail(P1,1)
tail(P1_scaled,1)
```

```
[1] 0.01675842
[1] 0.01675842
[1] 1.137895
[1] 1
```

```
P2 <- cumprod(1 + c(0, rnorm(20, sd = 0.02)))
P2_scaled <- scale1(P2, centre = TRUE, scale = 0.03)

sd(returns(P2))
sd(returns(P2_scaled))

tail(P2_scaled,1)
head(P2_scaled,1)

cor(returns(P1), returns(P2))
cor(returns(P1_scaled), returns(P2_scaled))
```

```
[1] 0.01948189
[1] 0.03
[1] 1
[1] 1
[1] 0.1147915
[1] 0.1147915
```

10 Plotting irregularly-spaced series during trading hours

10.1 An example

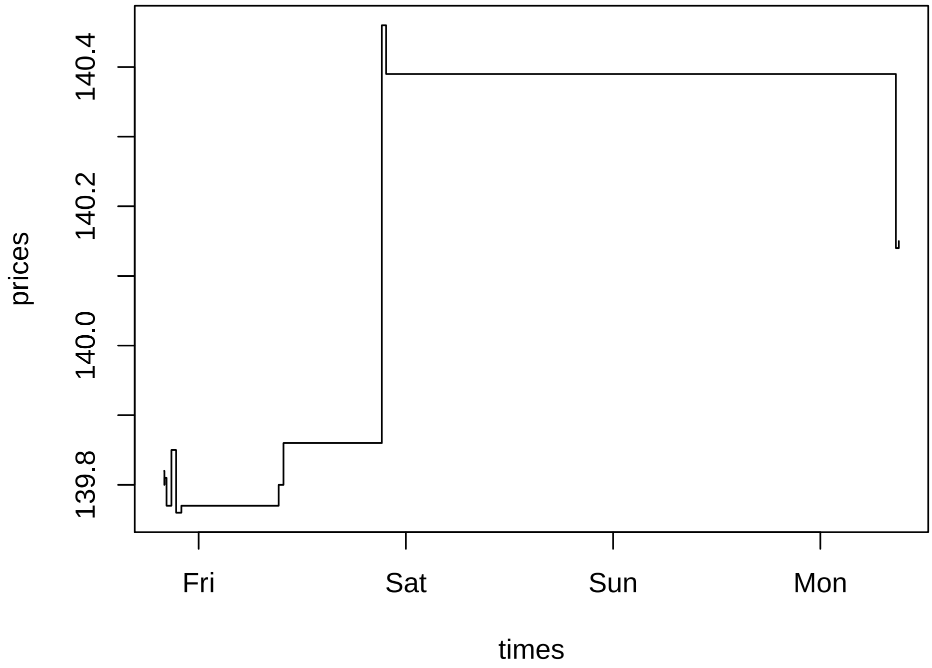
We have the following sample of prices of the Bund future contract, traded at the Eurex in Germany.

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

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` (`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.

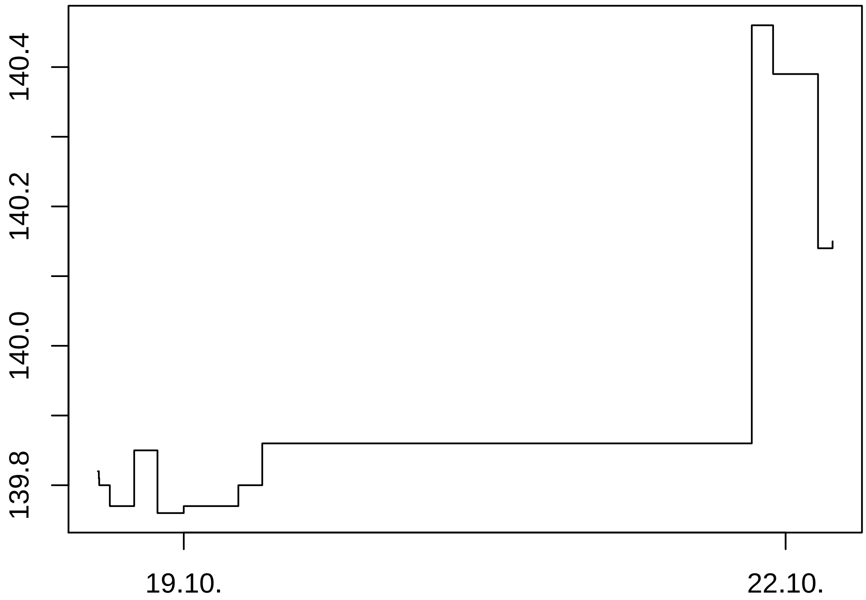
A `plot` of price against time looks like this.

```
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.

```
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:

day	x-position	mapped x-position
Thursday	1	1
Friday	2	2
Saturday	3	`<removed>`
Sunday	4	`<removed>`
Monday	5	3

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`).

```
str(tmp)
```

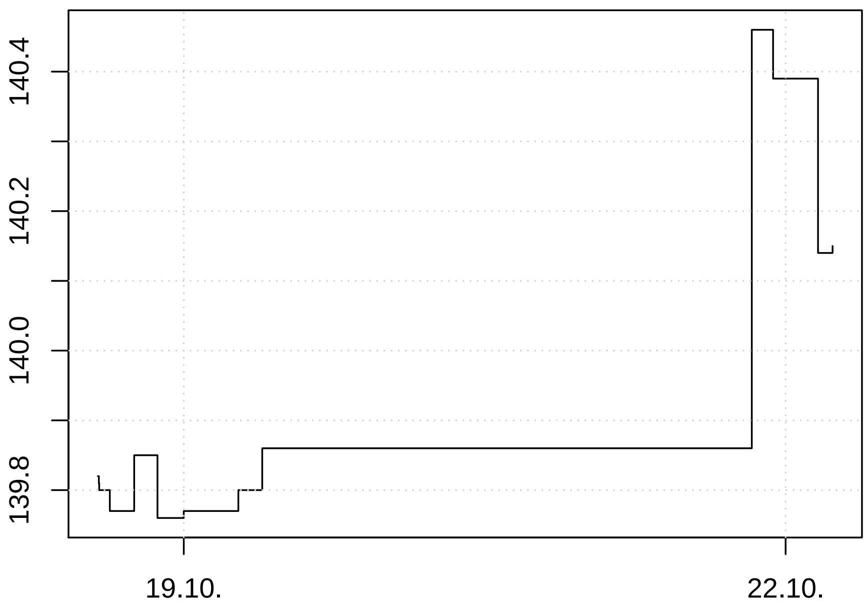
```
List of 6
 $ t          : int [1:15] 1 63 81 111 1001 3041 4991 6091 7191 11763 ...
 $ x          : num [1:15] 140 140 140 140 140 ...
 $ axis.pos   : num [1:2] 7193 57594
 $ axis.labels: chr [1:2] "19.10." "22.10."
 $ timegrid   : POSIXct[1:61527], format: "2012-10-18 20:00:09" ...
 $ map        : 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`.

```
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 wan 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`.

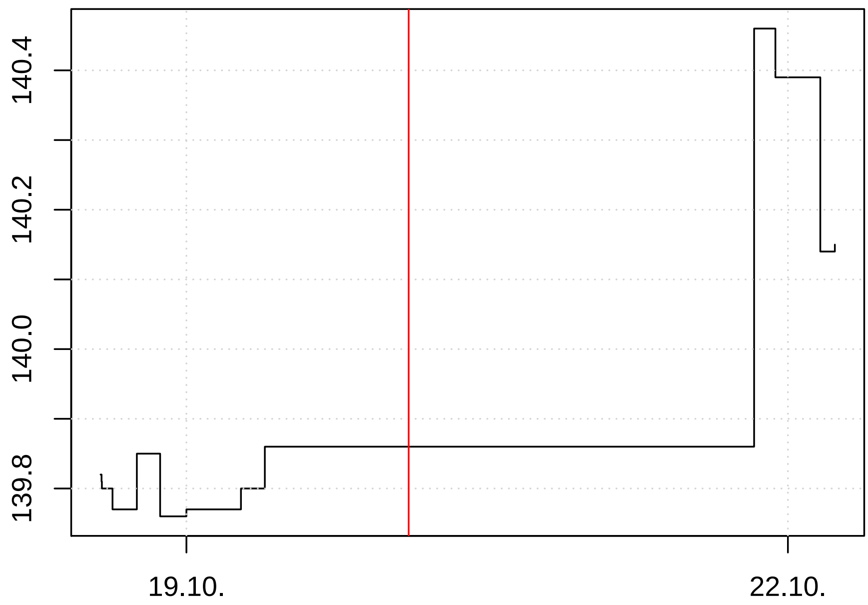
```
## Again, I do not specify a time zone since time zones
## depend on the operating system. To reproduce the
## example, you may use this representation:
##
## mytime <- structure(1350645023,
##                     class = c("POSIXct", "POSIXt"),
##                     tzone = "")

mytime <- ISOdatetime(2012, 10, 19, 13, 10, 23)
mytime
```

```
[1] "2012-10-19 13:10:23 CEST"
```

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

```
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$axis.pos,
       col = "lightgrey", lty = "dotted")
abline(v = tmp$map(mytime)$t, col = "red")
```



The function `map` returns a list with two components, `t` and `ix`.

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

```
$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.

```
## 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)
```

```
                times  weekday values
1 2012-10-18 21:00:00 Thursday  140.0
2 2012-10-19 21:00:00   Friday  140.1
3 2012-10-20 21:00:00 Saturday  140.2
```

But 20 October 2012 falls on a Saturday, and so it does not appear in the plot.

```
tmp$map(moretimes)
```

```
$t
[1]  3592 53993

$ix
[1] 1 2
```

The values that should be plotted can conveniently be found by using `ix`.

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

```
[1] 140.0 140.1
```

11 Other Tools

11.1 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`.

```
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.

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

```
[1] 9.086185 9.330603 9.207000 9.406000
```

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

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

```
[1]  9.77700 10.04000  9.90700 10.12113
```

11.2 Stocks splits

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

11.3 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` provides a way to 'pretty-print' such prices.

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

```
110-23+
110-23¾
```

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

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

```
[1] 110.7344 110.7422
```

11.4 Validating ISINs

An ISIN, which stands for International Securities Identification Number, uniquely identifies a security.

```
is_valid_ISIN(c("DE0007236101",     ## Siemens
                "DE0007236102"))    ## last digit changed
```

```
[1]  TRUE FALSE
```

11.5 Price tables

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

11.6 Trees

To *normal people*, a tree consists of a trunc, 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 hierachies – just think of a file tree.

12 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.
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, and make it a journal J_1, but {multiply all amounts by -1}. The prices need to be the valuation prices.
4. Combine J_0, J, and J_1 and compute the P/L.

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 `cumprod(c(1, 1 + r))`); then use `returns`.

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 `returns`: see arguments `weights` and `rebalance.when`. See Section Returns when weights are fixed.

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 `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?*

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.

13 Appendix: Classes and data structures

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

journal keeps transactions. Internally, a 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.

14 Appendix: Notes for developers

14.1 Methods for `returns`

Methods are responsible for ‘stripping’ the input down do `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.

15 Appendix: R and package versions used

```
R version 3.4.2 (2017-09-28)
Platform: x86_64-pc-linux-gnu (64-bit)
Running under: Ubuntu 17.04

Matrix products: default
BLAS: /usr/lib/openblas-base/libblas.so.3
LAPACK: /usr/lib/libopenblasp-r0.2.19.so

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

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

other attached packages:
[1] rbenchmark_1.0.0 zoo_1.7-14       orgutils_0.4-2
[4] PMwR_0.5-8

loaded via a namespace (and not attached):
[1] datetimeutils_0.2-7 compiler_3.4.2      parallel_3.4.2
[4] fastmatch_1.0-4     crayon_1.3.4        NMOF_1.2-0
[7] grid_3.4.2          textutils_0.1-8     lattice_0.20-35
```

Bibliography

[1] Dirk Eddelbuettel. *nanotime: Nanosecond-Resolution Time for R*. R package version 0.1.0. 2017. URL: `https://CRAN.R-project.org/package=nanotime`.

[2] Manfred Gilli, Dietmar Maringer, and Enrico Schumann. *Numerical Methods and Optimization in Finance*. Elsevier/Academic Press, 2011. URL: `http://nmof.net`.

[3] Enrico Schumann. *Numerical Methods and Optimization in Finance (NMOF) – Manual (Package version 1.1-0)*. 2011–2017. URL: `http://enricoschumann.net/NMOF.htm#NMOFmanual`.

Index