

# Backtesting\*

Enrico Schumann

es@enricoschumann.net

December 2018

## 1 What is (the problem with) backtesting?

Backtesting means simulating an investment strategy on past data. Most of the time, the word is used in the context of systematic or rule-based investment strategies. Such trading had often been called mechanical in the past, and today it goes by the name of automated, algorithmic or quant trading.

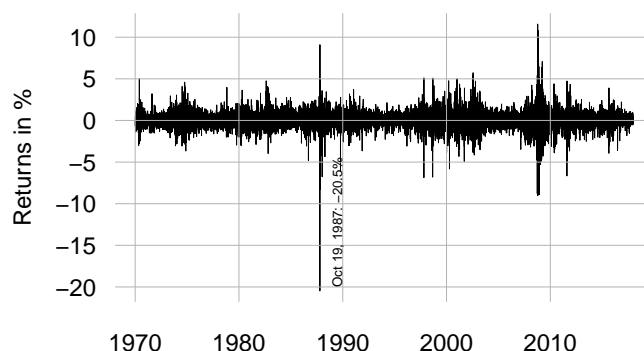
Simulating historical performance is, in principle, not limited to systematic strategies: if a discretionary trader has kept a diary of his trades, we might as well simulate his trading history. Or we may test the predictions that a newspaper has made in the past, and simulate the financial outcomes of its investment advice. There is, however, a difference between tracking a newspaper's recommendations and running a systematic strategy on past data: the first was live, though its performance has not been measured; the second was not. This chapter is about the second, non-live type. The software we describe may, however, be used for the first type as well.

Clearly, the types of trading that we mentioned encompass a wide range of strategies, from algorithms that operate at ultra-high frequency at the order book level, to fundamentals-based investment models that are rebalanced once per year. There is no way that we could treat them all in a single chapter (not to mention that one could not plausibly be knowledgeable enough in all those different types of trading); we shall thus limit ourselves to several examples, which will also give away the area in which the author of this chapter has been working.

---

\*This is a draft of a chapter for the second edition of "Numerical Methods and Optimization in Finance," by M. Gilli, D. Maringer and E. Schumann, to appear in 2019. Comments, suggestions and corrections are very welcome.

Figure 1: Daily returns of the S&P 500 between January 1970 and December 2017 (11,976 days). On October 19, 1987 the index dropped by more than 20%.



So here is the plan for the chapter. Throughout the first part, we will try to make the point that backtesting is difficult and perilous: preparing the inputs, running the backtest, and finally interpreting the results comes with pitfalls and problems. But the purpose of the chapter is not to discourage you from running backtests: backtesting is a necessary step to gain insight into a strategy, and it is an indispensable part of empirical work. Thus, the goal is not to deter, but to raise awareness of some of the pitfalls—and to suggest possible remedies. In the second part of the chapter, we will describe software for running backtests,<sup>1</sup> and illustrate the software through several examples from US equity markets.

But as we said, we start with the problems.

## 1.1 The ugly: intentional overfitting

In many areas of the financial world, the word backtest has a very bad reputation. The reason is simple: run enough backtests, and you are bound to find strategies that look good in-sample, i.e. on a particular data set. Even worse, the number of backtests you have to run for good results is surprisingly small.

Financial models are what numerical analysts call sensitive: small changes to the inputs of computations often lead to large changes in the results. A single day, for instance, can have a large impact even over a long time period:<sup>2</sup> Just think what a difference it makes to include or exclude October

<sup>1</sup>The description builds on Schumann (2008–2018a).

<sup>2</sup>See the examples in Gilli, Maringer, and Schumann, Chapter 1. Extreme events may be rare, but their impact is large, even when downweighted by their frequency. What is more, if such events affect returns, the impact will propagate because of the geometric chaining of returns.

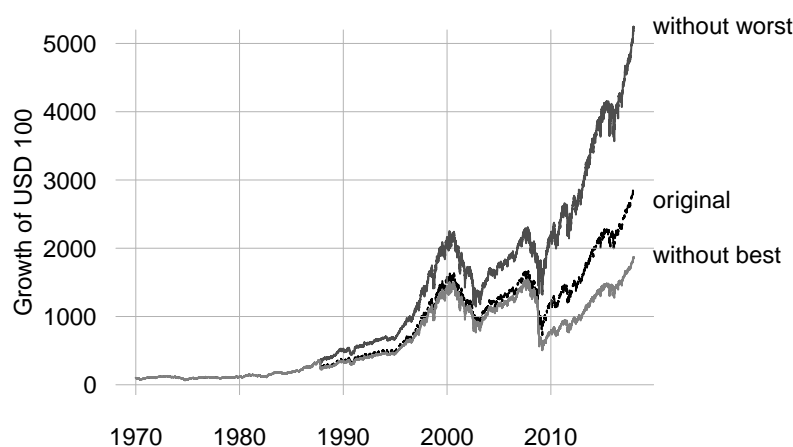


Figure 2: S&P 500 between January 1970 and December 2017 without 10 best and 10 worst days. Having missed the best 10 days in the market would have led to a cumulative underperformance of more than 50%, or 0.9% p.a., against the market. That is, an investor in the S&P would have ended with a wealth more than 50% higher. Avoiding the 10 worst days would have led to an outperformance of more than 80%, or 1.3% p.a., against the market.

19, 1987 into a study on equity market returns; see Figs 1 and 2.

This sensitivity implies that changes, even miniscule ones, to the configuration of a backtest often lead to meaningful changes in the results. Some such changes will, by mere chance, be positive. It is thus very likely that in the neighborhood of essentially any strategy, one can find even better configurations, making it tempting and easy to “improve” a strategy by changing parameters. There typically is no real improvement, of course; only improved statistics for a particular dataset. Nevertheless, it opens the door for fiddling with a backtest until it looks good. At the same time, it becomes more difficult to evaluate the robustness of strategies.

Let us make an experiment: run a strategy on random data. For this experiment we make use of packages `NMOF` (Schumann, 2011–2018) and `PMwR` (Schumann, 2008–2018b), which we first load and attach. We also set a seed to make the results reproducible.

```
> library("NMOF")
> library("PMwR")
> set.seed(2552551)
```

[seed]

We define a function `randomPriceSeries`, which creates a simple random walk by chaining together Gauss-distributed returns with zero mean.<sup>3</sup> The created series always has an initial value of 100. For intuition, think of daily observations of a stock price. (See the Gilli et al., Chapter 6 for how to create such series.)

<sup>3</sup>The function also has an argument `demean`: if set to true, the series will be scaled so that its total return is zero. You are encouraged to experiment with the code to see how different settings affect the results.

```
[rnd-series] > randomPriceSeries <-
              function(length, vol = 0.01, demean = FALSE) {

                x <- cumprod(1 + rnorm(length - 1, sd = vol))
                scale1(c(1, x), centre = demean, level = 100)

              }
```

The function allows us to define a return volatility (argument `vol`), with default 1%, which would translate into an annual volatility of 16%. We create and plot a sample path, shown in Fig. 3.

```
[rnd-series-fig] > x <- randomPriceSeries(250)
> plot(x, type = "s", xlab = "", ylab = "x", )
```

Actually, if you run the code as it is shown here, you will find that your plot looks slightly different from the one printed in Fig. 3. That is because before calling `plot`, we called `par` with several settings, collected in a list `par_btest`.

```
[par] > par_btest
```

```
$bty
[1] "n"

$las
[1] 1

$mar
[1] 3 3 1 1

$mgp
[1] 2.0 0.5 0.0

$tck
[1] 0.01

$ps
[1] 9
```

Before every plot in this chapter, we invoke

```
> do.call(par, par_btest)
```

(or a slight variation of it). In this way, we may reuse the settings for other graphics in this chapter. We typically omit the line in the shown code; you

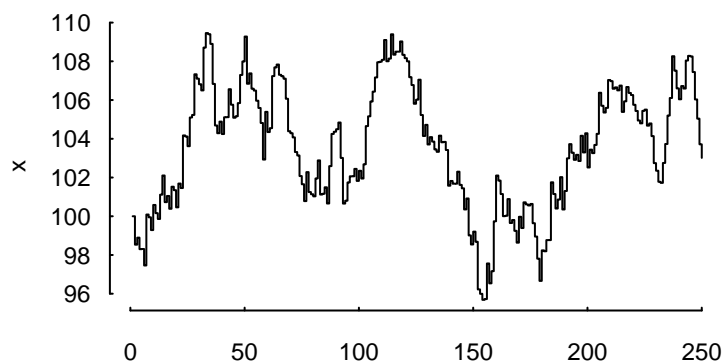


Figure 3: A random series created with function `randomPriceSeries`.

can, however, find it in the chapter's R file. Also, in this chapter we often plot different data in essentially the same way; so we will typically show the code the first time we plot, and then omit it later. Again, the complete code is in the chapter's R file.

Let us test a simple technical trading system on this random series. We compute two moving averages,  $m_{10}$  and  $m_{30}$ . The subscripts stand for the order of the averages (again, think of trading days). Whenever  $m_{10} > m_{30}$  we invest our total wealth in the asset; otherwise we do not hold a position and keep cash instead. The function `MA_crossover` implements the strategy and simulates its results on a series  $S$ .

```
> m <- c(10, 30) [ma-crossover]

> MA_crossover <- function(m, S) {
  m.fast <- MA(S, m[1], pad = NA)
  m.slow <- MA(S, m[2], pad = NA)

  crossover <- function() {
    if (m.fast[Time()] > m.slow[Time()])
      1
    else
      0
  }
  tail(btest(S, signal = crossover,
            b = 60, initial.cash = 100,
            convert.weights = TRUE)$wealth,
      n = 1)
}
```

`MA_crossover` first computes two moving averages. It then defines a new

function, `crossover`, and passes it to function `btest`. We shall explain later in this chapter what exactly the code does. It should suffice for now that it returns the final wealth of an investment that starts with an initial wealth of 100. Let us try the function with our random series.

```
> MA_crossover(m, x)
```

```
[1] 92.9
```

Let us try another series.

```
> x <- randomPriceSeries(250)
> MA_crossover(m, x)
```

```
[1] 93.5
```

At first sight it should not come as a surprise that the strategy did not work well: after all, the asset's price is random, and we know there is no exploitable pattern. (We did not deduct transaction costs.) But let us run the experiment 100 times, and see what happens.

[experiment1]

```
> m <- c(10, 30)
> initial_wealth <- 100
> buyhold_profit <- final_profit <- numeric(100)
> for (i in seq_along(final_profit)) {
  S <- randomPriceSeries(250)
  final_profit[i] <- MA_crossover(m, S) - initial_wealth
  buyhold_profit[i] <- S[length(S)] - S[1]
}
```

We store the final profit of all runs in a vector `final_profit`. The profits of simply holding the underlying random series are stored in `buyhold_profit`. To control for a rising underlier, and to see how the strategy fared, we summarize the differences in final profits, i.e. strategy profit minus buy-and-hold-profit. (Alternatively, we might have set `demean` to `TRUE` when we created the series.)

```
> summary(final_profit - buyhold_profit)
```

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
-29.3	-8.3	0.9	0.6	7.8	32.4

The following code block computes the distribution of the profit differences and plot it; see Fig. 4. Roughly half of the strategies outperformed the asset.

[rnd-profits]

```
> plot(ecdf(final_profit - buyhold_profit),
```

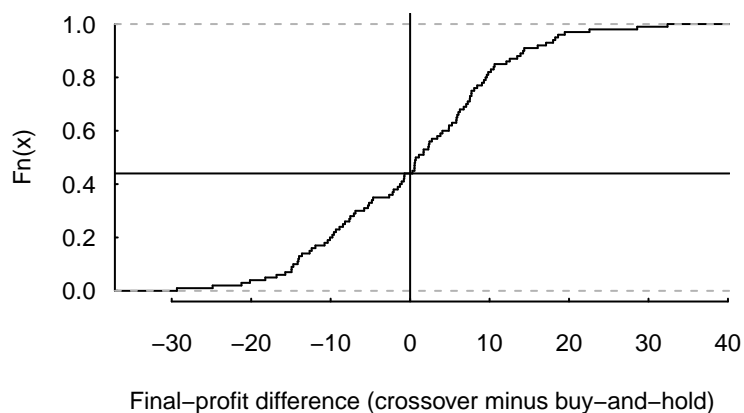


Figure 4: Distribution of difference in final profits MA-crossover strategy vs buy-and-hold. A positive number means that MA-crossover outperformed the underlying asset. The distribution is symmetric about zero, which indicates that there is no systematic advantage to the crossover strategy.

```
main = "",
pch = NA,
verticals = TRUE,
xlab = paste("Final-profit difference",
              "(crossover minus buy-and-hold)"))
> abline(v = 0,
          h = ecdf(final_profit - buyhold_profit)(0))
```

Fig. 5 shows a scatter plot of the buy-and-hold profits and the strategy profits. We use function `eqscplot` from package `MASS`, which automatically chooses equal scales on both axes.

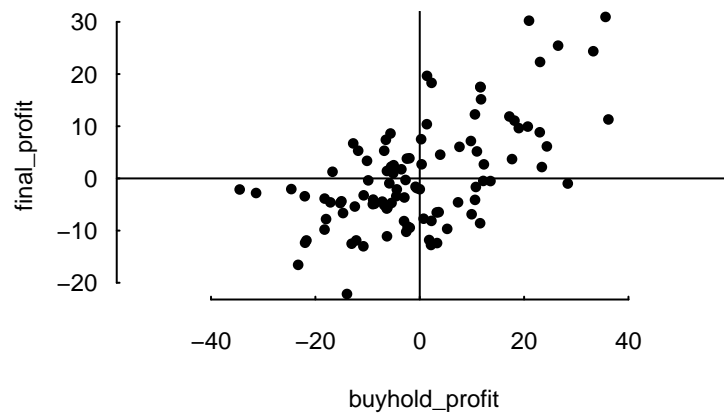
```
> library("MASS")
> do.call(par, par_btest)
> eqscplot(buyhold_profit, final_profit,
            main = "", pch = 19, cex = 0.6)
> abline(v = 0, h = 0)
```

[rnd-profits-corr]

In sum, running the crossover strategy on random data had about a 50/50 chance of outperforming the asset on which it was based.<sup>4</sup> When you reflect on the result, it may appear obvious: there is no systematic bias in the random underlying time series, and hence the crossover strategy could not pick up any advantage. But the purpose of the experiment was not to compute the average outcome. Instead, we wanted to show the range of possible outcomes: many realizations did poorly; but others did very well, merely because of chance. Imagine yourself in a world with tens of thousands of quant traders (quant monkeys) who test different strategies on different assets. Even after

<sup>4</sup>In real life, the chances may be lower because of transaction costs and because of the properties of a strategy: for instance, a strategy that is mostly short in a rising market will mostly lose out.

Figure 5: Final profits of MA-crossover strategy vs result of buy-and-hold of the same underlying series.



transaction costs and other hurdles, a large number of apparently successful strategies will remain.

Let us run a second experiment. Before, we used 10 and 30 as the parameters for the strategy. Let us see what happens when we choose the parameters in a more “optimal” way. We fix admissible ranges for the parameters: 1 to 20 for the fast moving average, and 21 to 60 for the slow one. Since we have only two parameters to check, we may run a backtest for each combination and keep the best backtest. One possibility to run such an optimization is through a nested loop: one loop for each parameter. The function `MA_crossover_optimized` will do that for us.

```
[ma-crossover-opt] > MA_crossover_optimized <- function(S) {
  fast <- 1:20
  slow <- 21:60

  crossover <- function() {
    if (m.fast[Time()] > m.slow[Time()])
      1
    else
      0
  }
  best <- -10000
  best.par <- c(0, 0)
  for (f in fast) {
    m.fast <- MA(S, f, pad = NA)
    for (s in slow) {
      m.slow <- MA(S, s, pad = NA)
      res <- btest(S, crossover, b = 60,
        initial.cash = 100,
```



```

        convert.weights = TRUE)
    if (tail(res$wealth,1) > best) {
        best <- tail(res$wealth,1)
        best.par <- c(f, s)
        best.wealth <- res$wealth
    }
}
}
attr(best, "wealth") <- best.wealth
attr(best, "parameters") <- best.par
best
}

```

Just like `MA_crossover` before, it returns the final profit for the given dataset; it also returns (as attributes) the equity curve of the strategy and the best parameters, which lead to that equity curve.

```

> x <- randomPriceSeries(1000)
> res <- MA_crossover_optimized(x)

```

Actually, the `NMOF` package offers a more convenient function for such a computation: `gridSearch`. `gridSearch` will minimize, so we need to flip the sign of the function we pass in. We can do this by wrapping `MA_crossover` into an anonymous function.

```

> res2 <- gridSearch(function(m, S)
                    -MA_crossover(m, S),
                    S = x,
                    levels = list(1:20, 21:60))

```

[grid-search]

Fig. 6 shows three examples of random paths and the equity curves of the associated optimized strategies. The graphics provide an indication of the typical results we get. But let us be a little more systematic and run the experiment 100 times (i.e. on 100 random paths). The variable `initial_wealth` remains 100, as defined before.

```

> buyhold_profit_opt <- final_profit_opt <- numeric(100)
> for (i in seq_along(final_profit_opt)) {
    x <- randomPriceSeries(1000)
    res_gs <- gridSearch(function(m, S) -MA_crossover(m, S),
                        S = x,
                        levels = list(1:20, 21:60))
    final_profit_opt[i] <- -res_gs$minfun - initial_wealth
}

```

[experiment2]

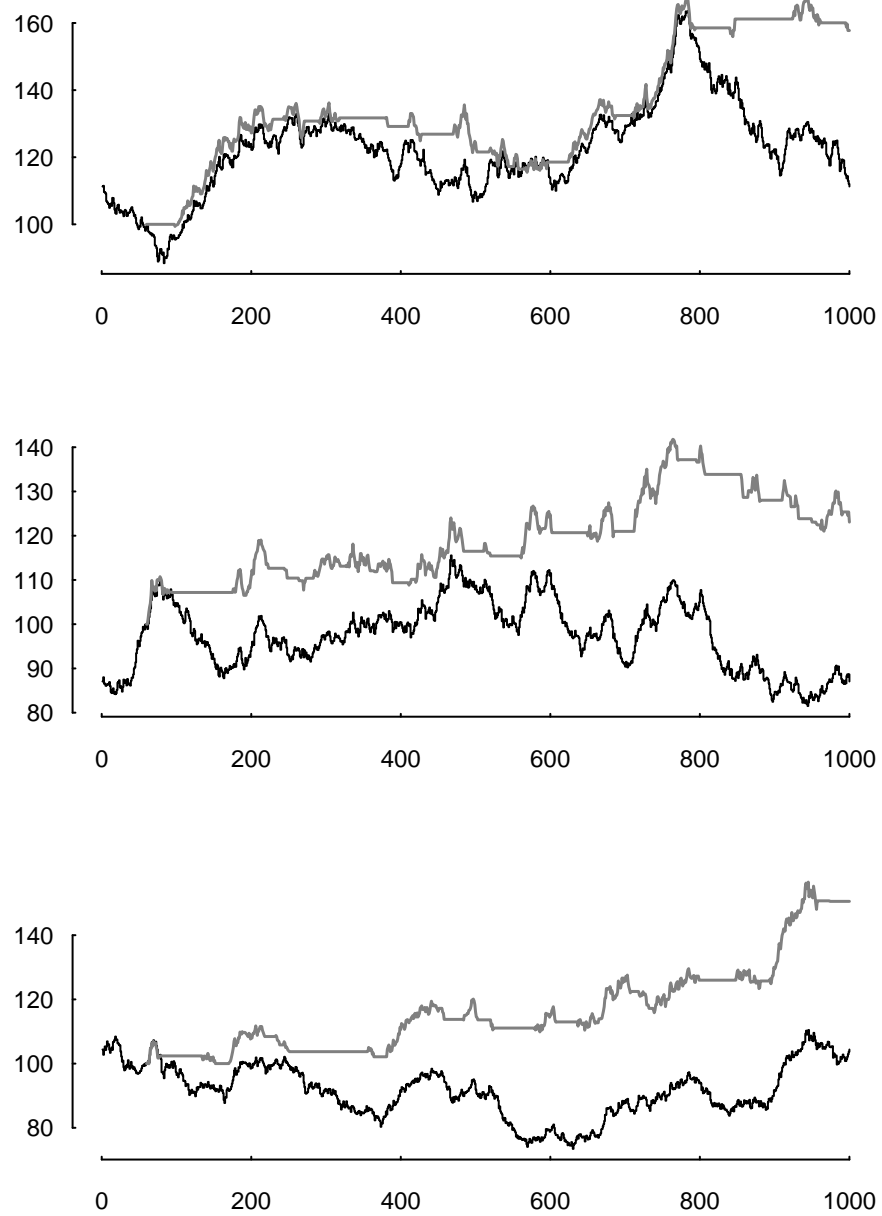


Figure 6: Optimized backtests. The black lines show the asset; the gray lines show the performance of the MA-crossover strategy with optimal parameters.

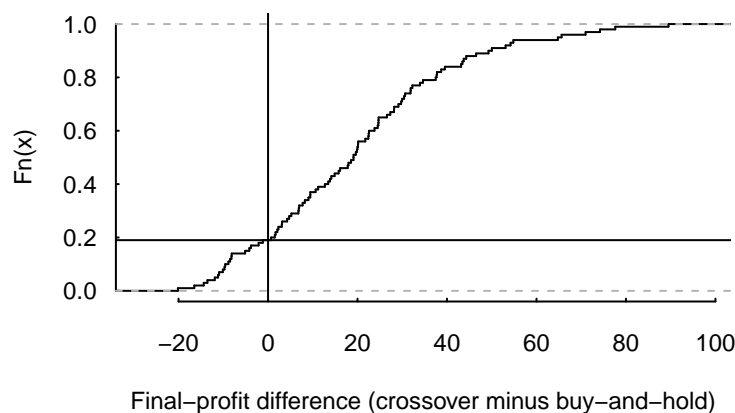


Figure 7: Distribution of in-sample excess profits for optimized MA crossover strategy. A positive number means that MA crossover outperformed the underlying asset.

Again, we summarize the results; a plot of the distribution of the differences in final profits is shown in Fig. 7.

```
> summary(final_profit_opt)
```

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
-20.1	3.1	19.1	19.8	31.9	89.5

Compare Fig. 7 with 4: only very few of the optimized strategies now perform worse than the asset. This fact – that even simple models with only few parameters may be fitted so that they perform well in-sample – has given rise to the bad reputation of backtesting. Because this fact it has been exploited to trick investors into supposedly brilliantly-performing strategies, when in truth they were nothing more than backfitting exercises.

Such intentional overfitting is not the purpose of this chapter; but we hope the experiment helps raise awareness. Despite this dark potential, backtesting is a highly useful technique. The remaining part of the chapter offers advice on how to use it.

## 1.2 The bad: unintentional overfitting and other difficulties

At the risk of repeating ourselves: Backtesting is difficult. There are many problems besides overfitting.

- In finance we have little data to learn from. That is at least when compared with the vast amounts of data that are available in other areas, e.g. the billions of photos that Google has at its disposal to train its face recognition methods, say, all neatly tagged so that algorithms

may learn to recognize cats or whatever. These kinds of data are simply not available in finance. It is true that once we move into high-frequency data, the size of datasets grows quickly. But this only “zooms into” existing data: having tick data for a year in which the average stock rose may mean many observations, but these are not independent of one another. The situation gets worse when we work with lower-frequency data types, such as fundamental or macroeconomic data.

- We require accurate historical data. For instance, we need to know what assets could be bought and sold at what prices; and we need to know how these assets performed during the past. For regulated markets in developed countries in recent years, such data are typically available.
- When we simulate a trading strategy, we may need to account for its influence on the market. On a high level, this may mean that we need to make assumptions on price impact. Or think of an extreme case: you want to test an algorithm that acts on patterns in the order book. It is much more difficult to test such an algorithm on historical data, because if the algorithm acts on what it sees, it will issue orders, and hence you start to change the historical track-record of the order book.

Once we have run backtests, we need to analyze their results. Suppose we ran tests that actually showed that the strategy we are interested in was profitable. Then there are still several points to keep in mind.

Any test depends on time and assets: clearly, in a equity bull market it should not be surprising that a strategy with a long bias performs well. It helps to use a realistic benchmark; but it is often hard to find such a benchmark. This is particularly a problem when we work with a strategy that acts on a large cross-section of assets. Suppose we have run a cross-sectional equity strategy and find that it outperformed the benchmark. It may only have been because it picked up certain factor exposures: the classic market factor, or Fama–French factors (value, small cap) are candidates to test. But the burden of coming up with relevant benchmarks or risk factors lies with the Analyst. For single-instrument strategies, similar problems arise, notably related to exposure: it becomes much more difficult to evaluate a strategy that is often not invested, or changes leverage frequently.

As we pointed out above, financial models are sensitive: small changes may lead to very large differences in performance. A single trade done or not done may make much of a difference. But that makes it difficult to judge whether a strategy’s profits were real or just luck. (Some authors have thus suggested to measure only cross-sectional improvements versus a benchmark, because there are many more bets than with a general timing strategy. But for one, this requires independent bets, which cross-sectional bets often are not; also,

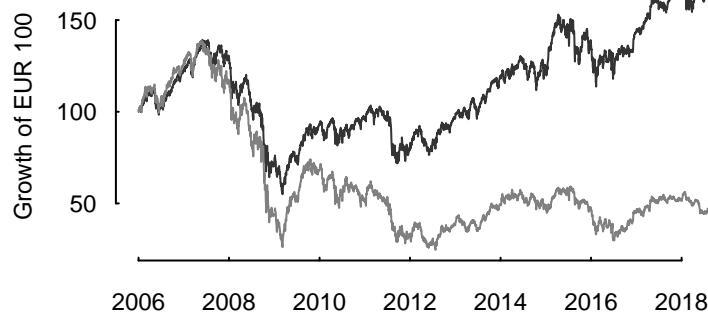


Figure 8: Performance of EURO STOXX Total Market and EURO STOXX Banks indices between January 2006 and July 2018. The weak performance of European banks is apparent. Many different models, such as momentum or low volatility, might have captured this trend.

just because we cannot measure well a quantity should not mean we may not try it.)

This sensitivity also makes it difficult to compare different strategies. We may have an observational equivalence between different strategies: different models may capture a single market trend. See Fig. 8 for an example. For a given sample, it is then impossible to differentiate between these models (based on the data alone), and hence, it is difficult to rank models.

That leaves us with unintentional overfitting. We may do it with the best of intentions, but: whenever we run many backtests on a single dataset, we are bound to find some that work, but which may quite likely be the result of data-snooping. What can be done? As long as we run tests on a single dataset, there is unfortunately little hard advice that can be given.<sup>5</sup> But the situation is not hopeless, and some informal rules do help:

---

<sup>5</sup>A number of papers have aimed to provide statistics to be adapted to multiple backtests, often following ideas from the statistical literature on multiple comparisons. The fundamental ideas of these papers is sound: after having run multiple backtests, one must not trust the results of a single backtest any more. Such papers are sometimes instructive, e.g. they show how very few backtests are needed to obtain good results, or when it is recommended that performance statistics be discounted. But some of the suggested statistics are harmful. For one, they provide a sense of exactitude that is not warranted. In particular, they are usually based on sampling backtests on a given data set, so they do not cover the case of iteratively “optimizing” a backtest. But there is an even worse aspect. A simple rule for evaluating strategies is that if a backtest looks too good to be true, it probably is not true. But some suggestions in the literature (e.g. Harvey, Liu, and Zhu, 2016) do the contrary, actually: the authors’ solution is to insist on a higher level of statistical significance than normally used. But if we assume that useful strategies do not provide outlandishly-high Sharpe ratios, the proposed statistic would never flag a realistic strategy as a good strategy. On the other hand, a strategy that comes with a Sharpe ratio of 10, say, will likely pass the test, because of its high statistical significance (the Sharpe ratio is equivalent to a  $t$ -statistic).

- Try few things only: restrict yourself to few variants of a backtest, if at all. In any case, document everything you try, and read and reread those notes. (But do not use formal statistics such as ratios of good strategies to tested strategies or similar.) What is more, backtests should be motivated by observations in the market, or theory, but not by other backtests (“this did not work, but what if I changed this parameter ...”).
- Do not just analyze the performance of strategies, but see why they performed well or not in particular times or markets. It may be easier to evaluate a strategy when you consider the cause for its good performance, and in particular whether this cause will persist (see Fig. 8 again) or not.
- Be careful with composite strategies whose single components do not work on their own.
- Test the sensitivity of strategies: the goal is not to find a best variant of one model, but a model that works OK-ish in many different variants. See the examples below.
- Finally, or perhaps firstly: check your code. For instance, run it with random data to see whether any part can “see the future”.

Beyond working on a single dataset, there are two useful techniques: replication and meta analysis. Start with replication: Unfortunately, in the academic world, there is little incentive, because replication studies may not be easy to publish (unless you show that some seminal paper cannot be replicated). But replicating a backtest with new data, and perhaps also with a different implementation, can provide evidence for and confidence in a strategy. Meta analysis is related to replication, but broader. It might be summarized as “do not trust a single paper, or a single methodology.” Rather, collect published papers (and perhaps also unpublished ones) on a certain topic, and look at the range of outcomes. Again, in finance, this is made difficult because negative results (“the strategy did not perform well”) are rarely published.

And there is more, once we move beyond pure backtesting. We can set up our software for analyzing strategies so that data are updated as time progresses. That is good idea in any case, because it keeps the backtesting workflow close to production. And we may run real-time experiments: either in the form of paper trading, or real money trading. We may then follow the statistics of a strategy. This is similar to the clinical testing that is done in the pharmaceuticals industry. Of course, if you were interested only in definitive answers, we would need decades of live data. But we will never have definitive answers. Such experiments may nevertheless increment our understanding

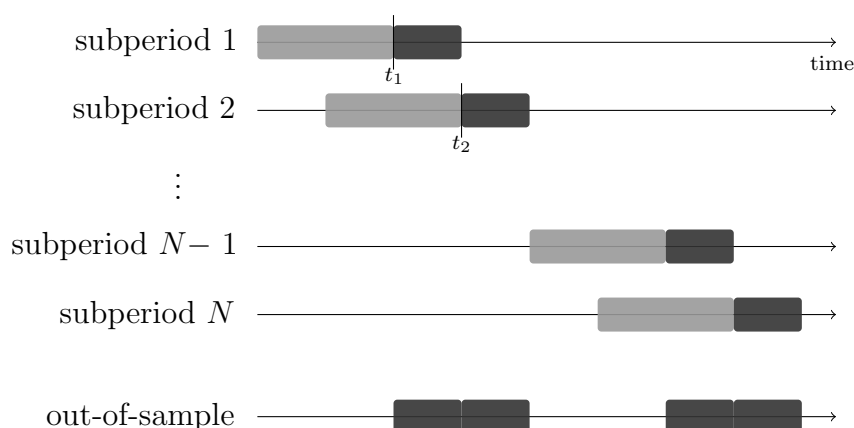


Figure 9: Walk-forward. The total time period is split into subperiods 1 to  $N$ , and each subperiod is split into an in-sample part (light-gray) and an out-of-sample part (dark-gray). The in-sample part may be used for determining trading parameters. In the end, the out-of-sample parts of all subperiods are concatenated.

of and confidence in a strategy. And one final remark on overfitting: grossly overfit strategies are usually discovered quickly, as their performance breaks down quickly when used on new data.

Altogether, there is a key insight to be taken from the discussion: in-sample results are not just dubious in financial backtests; they are typically worthless. This means that all backtesting must include some type of out-of-sample analysis. Of course, if you have followed the examples: even such out-of-sample tests, if done repetitively, are not going to save us. But they do help. Single splits of a dataset are typically not enough. Instead, for backtests the standard tool is a walk-forward. In a walk-forward, we split the dataset into many periods, each with one in-sample and one out-of-sample part. We may object at once: since we work on a historical dataset, there cannot be truly out-of-sample data. But the key idea is that for each subperiod, only the in-sample part may be used for determining the trading parameters. The procedure is summarized in Fig. 9.

Walk-forwards are not free of problems: the in-sample periods are overlapping, and hence different periods may be affected by a single event. (Out-of-sample periods never overlap.) And again, running many walk-forwards is data-snooping just as any other method. But walk-forwards give more reliable results, simply because a single method has been applied several times. And as you will see in the examples later in this chapter, such walk-forwards will (and should) always be accompanied by robustness checks.

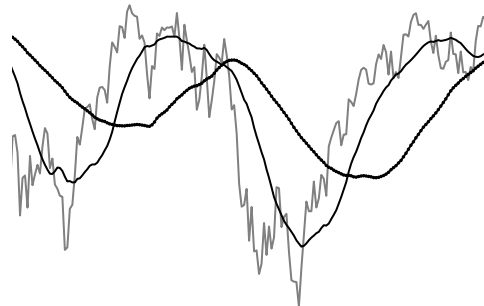
### 1.3 The good: getting insights (and confidence) in strategies

The discussion so far may have left the impression that backtesting is a dangerous undertaking. That is true; backtesting is difficult. But that does not mean one should despair.

As we said, there are remedies that may at least mitigate the troubles, such as careful data preparation; running only few tests; sensitivity checks; replication and meta analyses. In any case, no matter whether you judge other people's strategies or your own, stay skeptical and keep reasonable expectations: When people present strategies with outlandishly-good results (high returns, no drawdowns, etc.), or weird parameter values, be cautious. Always compare a strategy to what would be possible: Historically, equity markets had reward to risk of perhaps 0.3. It is simply not probable that some equity strategy comes with an expected Sharpe ratio of 6.

With all that said: backtesting is a wonderful tool. It is the only way to make empirical statements about strategies. Just listen to popular investment advice on TV, or read it in a newspaper: these are full of statements such as "a company so cheap must be bought", or "no company should have so much debt", or "buying after the market declined is always a good idea". Essentially none of such statements are empirically verified or even scrutinized. There is no doubt that looking at past performance gives no guarantee for future returns. But when a claim is made that some strategy has worked, and tests show that it has not, quite a bit has been learnt. So do test strategies. As argued in Chapter 1 of Gilli et al., finance does not suffer from too much empirical research, but from too little. Backtesting is much preferred over anecdotal evidence.

Backtests may also help against visual traps and selective perception. A good example is a simple strategy of moving averages, just like the one we used in the initial experiments. Any book on technical analysis is going to show you a graphic like the one the right. But such (carefully chosen) graphics are often much more promising than the results of a systematic test of such an indicator, which would be done via a backtest.



Also, backtesting is preferable to only running regressions or computing other summary statistics as proxies for potential profits. Running a real



backtest will make sure all units are right, trade logic is followed, and there are no forgotten assumptions. It is much easier to make mistakes when we look only at correlations, say, or by proxying potential profits from a strategy by chaining together returns. See Blume and Stambaugh (1983) for a well-known example of the latter problem, which would not occur with an actual backtest.

## 2 Data and software

Let us turn to the implementation of backtesting software.

### 2.1 What data to use?

Backtests necessarily work on past data. It is important that the data do not introduce biases, such as lookahead.

There is a simple principle regarding what information and data can be used: at any point in time, the algorithm must only rely on data and information that actually was available. Likewise, it must only invest in securities or assets that could have been traded, and only at prices at which these assets were tradable.<sup>6</sup>

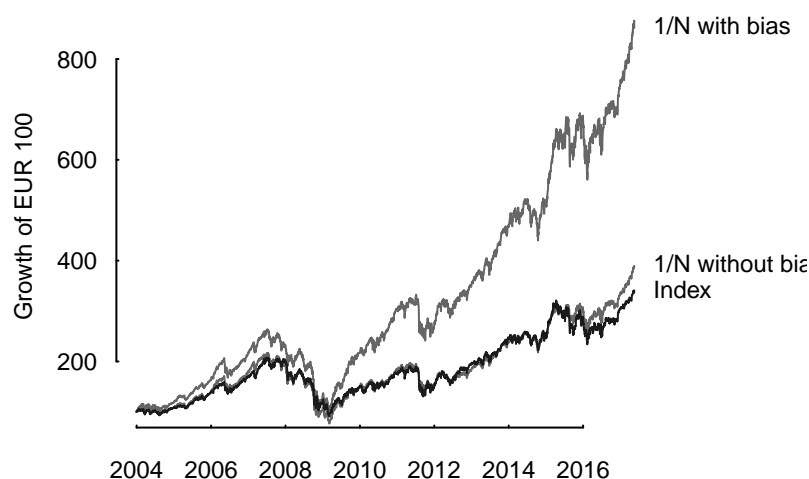
This principle may be obvious enough, but it is not necessarily easy to follow. Start with *what data* were available. Many databases store the actual timestamps of data, but not when the data became available. Think of fundamental data: suppose the earnings of a company are dated December 31 of a given year; yet they were certainly not available on that date. The company may have reported the numbers on March 12 the following year, say. Unless you followed the company directly, but used data provider instead, it may still have taken several more days before the data were entered into the provider's database. To make matters worse, such reports may have been restated later, and the provider may only have stored the most recent (restated) version.

A less obvious problem is the question *what assets* were available. Suppose we want to test a strategy in U.S. stocks, and we define our universe to be that of the S&P 500. We cannot simply fetch data on the current components of the index and test our strategy: over time, companies enter and exit the index, but these changes are not random. Badly performing companies (which

---

<sup>6</sup>An example that often comes up are futures prices. Futures have a limited lifespan, often only a few months. The cleanest method is, per the stated principle, to rebalance into the active futures contracts, i.e. to roll over a position. To simplify computations, continuous series are often used, which is fine for many strategies, given that they are chained together properly (e.g. by using returns).

Figure 10: Gilli and Schumann (2017) study strategies that invest in German equities over the period from January 2003 to May 2017. Their dataset comprises the components of the so-called HDAX, which bundles the 110 largest stocks in Germany. Gilli and Schumann (2017) find that when correcting for survivorship bias, an equally-weighted portfolio of the components of the HDAX would have performed similarly to the index, as shown in the figure. However, using only the most recent components would have lead to a survivorship bias of about 7% per year.



includes those that go bankrupt) are excluded, and highflyers enter. See also Bessembinder (forthcoming), who reports that for the United States the median life-time performance of stocks is worse than that of treasury bills. Only looking at the current components of an index thus would introduce a survivorship bias into our analysis. This bias may not only affect aggregate returns; it may also overstate the performance of particular strategies, such as buying-the-dips or momentum. This particular bias has been studied in Daniel, Sornette, and Wöhrmann (2009), and they found that the bias can be huge, with up to 8% higher in-sample performance of strategies that suffered from the bias. Gilli and Schumann (2017) ran tests for the German equity market over the period 2003–17 and found a bias of similar magnitude; see Figure 10.

A second problem related to the availability of assets, and perhaps a more obvious one, is volume. A strategy that invests in small caps, for instance, must make allowance for illiquid stocks. Short sales may not have been possible or may have been very expensive. Also, reported prices were not necessarily tradeable: bid–ask spreads, for instance, may or may not be reflected in reported prices. But either case may need treatment. If only the midprice is reported and used, we understate transaction costs. On the other hand, if reported prices are simply last prices, they may reflect bid or ask prices, and hence an algorithm may be inclined to learn to buy at the bid and sell at the ask. Settlement prices also may require attention: depending on the exchange, these may only be computed, i.e. do not constitute tradeable prices

(e.g., for German Bund futures, the settlement price reported by the Eurex is actually an average price). If in doubt, one may have to check the exchange procedures.

In sum, running backtests requires careful data preparation, which typically takes much time.

## 2.2 Designing backtesting software

Writing software for backtesting depends on what you want to test. There are some obvious requirements: the software should take care of repeated tasks, such as the accounting of trades, and it should allow reporting. Beyond that, much is left to the preferences or requirements of the user and the particular strategies, and there are different tradeoffs.

- flexibility versus convenience: software may rely on building blocks, such as strategy components, which make writing composite strategies faster, though sometimes at the price of being less flexible
- reusability of code: ideally, we could use the same code in testing and in production. Depending on the overall setup, this may come at the price that quickly testing a strategy becomes more difficult.
- simplicity: are we simply testing broad ideas, or do we need the fine details of trades, down to modeling the order book. Again, there is a trade-off, as more details typically mean that it takes more time to implement a strategy.
- speed: do we want to use the code as input to an optimization routine? If yes, we may have to trade off convenience for speed.

In any case, we stress the idea of backtesting software, not just algorithms. Backtesting software is a great example for code reuse: using the same implementation for different strategies saves time, and errors that are found can be fixed everywhere at once.

## 2.3 The `btest` function

In this section, we give a high-level introduction to one possible implementation of a backtesting software. This implementation can be found in the function `btest` in package `PMwR`, which we already used. In terms of the trade-offs described in the previous section, `btest` is strongly biased towards flexibility and simplicity.

The logic of `btest` can be summarised via two questions, which at a given instant in time (in actual life, “now”), a trader needs to answer:

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.

Implementing `btest` required a number of decisions 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 vectorize instead? Vectorization 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. 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. Thus, `btest` relies on a loop internally. To implement the questions above, it relies on a functional approach.

In fact, 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. You will see examples in the following section. Using functions in this way 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, as described 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.

### 3 Simple backtests

(Simple means univariate.)

### 3.1 btest: a tutorial

When it comes to computation, much of backtesting is accounting: keeping track of what is bought and sold at what time, and valuing open positions. This accounting we shall leave to the function `btest`, which we describe in this section. The function is provided in the R package `PMwR`, which we load and attach first.

```
> library("PMwR")
```

#### A useless first example

We really like simple examples. Suppose we have a single instrument, and we use only its closing prices. Let us make up some prices.<sup>7</sup>

```
> prices <- 101:110
> prices
```

```
[1] 101 102 103 104 105 106 107 108 109 110
```

The trading rule is buy one unit of the asset and hold it forever. (As we said, it is to be a simple example.) Here is how we could test this strategy with `btest`.

```
> bt.results <- btest(prices, function() 1)
> bt.results
```

```
initial wealth 0  =>  final wealth  8
```

Great: we increased our wealth from zero to 8. (Which is a lucky number in China.) But perhaps we had better explain what the code did. Let us start with the input.

The function first takes the prices as an argument. The prices should be ordered in time, and they will be used for determining entry and exit prices, and also for valuing open positions. `btest` actually expects open, high, low and close prices, but it can work with close prices alone as well. See the appendix on page 75.

As a second argument, we pass a function, which simply returns 1 whenever it is called. Note that we wrote it as an anonymous function, because it was so short. But we could as well have defined a named function:

```
> fun <- function()
```

---

<sup>7</sup>Such prices are genuinely useful when you debug code.

```

1
> bt.results <- btest(prices, fun)

```

This function that we pass to `btest` is called the `signal` function. It must be written so that it returns the target, or suggested, position, in units of the asset. The position is only suggested because we may, for some reason, prefer not to trade, so this suggested position never becomes an actual one. In this first example, the suggested position always is 1 unit. Under the default settings, the `signal` function is called at every instance in time. Since we did not explicitly define time, `btest` loops over the prices and calls `signal` for every price.

To make the description more precise, let  $T$  be the number of price observations, i.e. here the length of `prices`. Then `btest` function runs a loop 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, notably when our signals are based on returns, which is why the default burn-in is 1. In rare cases (such as our example)  $b$  may be zero. More often, it will be larger; for instance, when a strategy is based on a moving average.

Here is an important default: at time  $t$ , we can use information up to time  $t-1$ . Suppose that  $t$  were 4, i.e. the loop is at the 4th price observation. 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` (notably when we have no open price, as in our example):

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, and to sell at the low. Robustness checks—forcing 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.

With the inputs and the general workings described, let us show a little more output. The result of calling `btest` is a list of several components, notably `wealth` (the equity curve) and `position` (a matrix of the positions that the strategy has held). These components are by default not printed. Instead, a simple `print` method displays some information about the results:

```
initial wealth 0  =>  final wealth  8
```

In this case, it tells us that the total equity of the strategy increased from 0 to 8. The function `trade_details` extracts data from `bt.results` and prints them as a table.

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

	price	suggest	position	wealth	cash
1	101	0	0	0	0
2	102	1	1	0	-102
3	103	1	1	1	-102
4	104	1	1	2	-102
5	105	1	1	3	-102
6	106	1	1	4	-102
7	107	1	1	5	-102
8	108	1	1	6	-102
9	109	1	1	7	-102
10	110	1	1	8	-102

The initial cash is zero per default, so initial wealth is also zero in this case. (We may change it through the argument `initial.cash`.) As the table shows, we bought one unit in the second period at a price of 102. This position was then held until the final period, when it was valued at 110, leading to a final wealth of 8.

That we only bought at the second price is a result of the burn-in argument `b`, which defaults to one, and so we lose one observation. In the particular case here, we do not rely on the past, and so we may set `b` to zero. We now buy at the first price and hold until the end of time (i.e. the end of data).

```
> bt.results0 <- btest(prices = prices,
                      signal = function() 1,
```

```
b = 0)
> trade_details(bt.results0, prices)
```

	price	suggest	position	wealth	cash
1	101	1	1	0	-101
2	102	1	1	1	-101
3	103	1	1	2	-101
4	104	1	1	3	-101
5	105	1	1	4	-101
6	106	1	1	5	-101
7	107	1	1	6	-101
8	108	1	1	7	-101
9	109	1	1	8	-101
10	110	1	1	9	-101

Note that now we explicitly named the arguments in the function call. **btest** has more than 20 arguments, though almost all are optional. So naming arguments is the preferred way to call the function.

As we said, **btest** returns a list of several components, one which is a journal of the trades. It may be extracted with the function of the same name, **journal**.

```
> journal(bt.results)
```

	instrument	timestamp	amount	price
1	asset 1	2	1	102

1 transaction

```
> journal(bt.results0)
```

	instrument	timestamp	amount	price
1	asset 1	1	1	101

1 transaction

We will see later how to make the journal more informative by passing **timestamp** and **instrument** information. Similarly, you may extract the position by calling the function **position**.

```
> position(bt.results)
```

```
[,1]
```



```
[1,]    0
[2,]    1
[3,]    1
[4,]    1
[5,]    1
[6,]    1
[7,]    1
[8,]    1
[9,]    1
[10,]   1
```

Before we move on, a remark on data frequency. We have not made any specific assumption about data frequency: that is because the code makes none, so any frequency (intraday, daily, monthly, ...) 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 the user's responsibility to write `signal`—and other functions—in such a way that they encode a meaningful trade logic.

What other functions? When `btest` runs its loop along the prices, at every iteration, it goes back to the two questions we stated above: 1) Do I want to compute a target portfolio? If yes, do it. 2) Do I want to close the gap between the actual portfolio and the target portfolio? If yes, rebalance. The answers to these questions come from the functions `signal`, `do.signal`, and `do.rebalance`.

## Function arguments

We have already seen `signal`: it 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 a `signal` function that returns weights, set `convert.weights` to `TRUE`. Then, the value returned by `signal` will be interpreted as weights and will be automatically converted to position sizes.

The function `do.signal` is supposed to answer the first part of the first question: should we compute a signal at all? `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 should only be done at specific points in time. If the function is not specified, it defaults to `function() TRUE`. Instead of a function, `do.signal` 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
- a keyword such as `firstofmonth` or `lastofmonth` (in this case, `timestamp` must inherit from `Date` or be coercible to `Date`).

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

For completeness's sake: `btest` takes two more functions as inputs. One is `print.info`, which 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. The other function is `cashflow`. It is also called at the end of each iteration; its value is added to the `cash` position. The function provides a clean way to add accrued interest to or subtract fees from the cash account.

The default is to specify no arguments to these functions. But we see next that they may, nevertheless, access useful information.

### More-useful examples

Let us make the strategy more selective. The trading rule is to have a position of 1 unit of the asset whenever the last observed price is no higher than 105, and to have no position otherwise.

```
> signal <- function() {
  if (Close() <= 105)
    1
  else
    0
}
> trade_details(btest(prices, signal), prices)
```

	price	suggest	position	wealth	cash
1	101	0	0	0	0
2	102	1	1	0	-102
3	103	1	1	1	-102

4	104	1	1	2	-102
5	105	1	1	3	-102
6	106	1	1	4	-102
7	107	0	0	5	5
8	108	0	0	5	5
9	109	0	0	5	5
10	110	0	0	5	5

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

```
> signal <- function()
  Close() <= 105
> trade_details(btest(prices, signal), prices)
```

	price	suggest	position	wealth	cash
1	101	0	0	0	0
2	102	1	1	0	-102
3	103	1	1	1	-102
4	104	1	1	2	-102
5	105	1	1	3	-102
6	106	1	1	4	-102
7	107	0	0	5	5
8	108	0	0	5	5
9	109	0	0	5	5
10	110	0	0	5	5

The logical value of the comparison `Close() <= 105` would be converted to either 0 or 1. But the more verbose version above is clearer.<sup>8</sup> In the example, we—apparently—use the close price, but we do not access the data directly. Instead, we call a function `Close`. This function is defined by `btest` and passed as an argument to `signal`. Note that we do not add `Close` as a formal argument to `signal`; it is done automatically. In fact, adding it would trigger an error message:

```
> btest(prices, function(Close = NA) 1)
```

```
Error in btest(prices, function(Close = NA) 1) :
  'Close' cannot be used as an argument name for 'signal'
```

There is not only `Close` that can be accessed within `signal`. We also have

---

<sup>8</sup>To cite Brian Kernighan: “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?”

these objects:

**Open** open prices

**High** high prices

**Low** low prices

**Close** close prices

**Wealth** the total wealth (value of positions plus cash) 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

All these objects, with the exception of `Globals`, are functions. `Globals` is an environment, which can be used for storing data persistently. The listed functions may be called from within `signal`, and also from within the other functional arguments to `btest` (`do.signal`, `do.rebalance`, and so on). Because of that we call them inner functions. Inner functions can only read; there are no replacement functions, so you cannot change the close prices, for instance.

All inner 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 be decreasing. If you find it awkward to specify the lag in reverse order, you can 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. The function `Time` is particularly useful when it comes to accessing other data. Suppose our strategy were based not only on the last close price, but on a moving average. Specifically, let us buy when the last close is above the average of the previous  $k$  prices. Let us fix  $k$  at 3, which means we also need to set  $b$  to 3.

```
> signal <- function() {  
  k <- 3  
  ma <- sum(Close(n = k))/k  
  if (Close() > ma)  
    1  
  else  
    0  
}  
> trade_details(btest(prices = prices,  
  signal = signal,  
  b = 3),  
  prices)
```

	price	suggest	position	wealth	cash
1	101	0	NA	NA	0
2	102	0	NA	NA	0
3	103	0	0	0	0
4	104	1	1	0	-104
5	105	1	1	1	-104
6	106	1	1	2	-104
7	107	1	1	3	-104
8	108	1	1	4	-104
9	109	1	1	5	-104
10	110	1	1	6	-104

This implementation is straightforward, but it may be improved in two ways. First, the value of  $k$ . Hard-coding a variable in the code is bad practice. (Just as it was bad practice above to hard-code the price level of 105 within `signal`.) Instead we should add the parameter as an argument to `signal`, whose value we pass via the `...` argument of `btest`. Thus, we need to name the argument in the function call.

```

> signal <- function(k) {
  ma <- sum(Close(n = k))/k
  if (Close() > ma)
    1
  else
    0
}
> trade_details(btest(prices, signal, b = 3, k = 3), prices)

```

	price	suggest	position	wealth	cash
1	101	0	NA	NA	0
2	102	0	NA	NA	0
3	103	0	0	0	0
4	104	1	1	0	-104
5	105	1	1	1	-104
6	106	1	1	2	-104
7	107	1	1	3	-104
8	108	1	1	4	-104
9	109	1	1	5	-104
10	110	1	1	6	-104

```

> trade_details(btest(prices, signal, b = 5, k = 5), prices)

```

	price	suggest	position	wealth	cash
1	101	0	NA	NA	0
2	102	0	NA	NA	0
3	103	0	NA	NA	0
4	104	0	NA	NA	0
5	105	0	0	0	0
6	106	1	1	0	-106
7	107	1	1	1	-106
8	108	1	1	2	-106
9	109	1	1	3	-106
10	110	1	1	4	-106

The second improvement is in favour of speed. Since we know all prices, we may precompute the moving average and pass it as well. Note how we use `Time()` to access the current value of `ma`.

```

> ma <- MA(prices, 3, pad = NA)
> signal <- function(ma) {
  if (Close() > ma[Time()])

```

```

        1
      else
        0
    }
> trade_details(btest(prices, signal, b = 3, ma = ma), prices)

```

	price	suggest	position	wealth	cash
1	101	0	NA	NA	0
2	102	0	NA	NA	0
3	103	0	0	0	0
4	104	1	1	0	-104
5	105	1	1	1	-104
6	106	1	1	2	-104
7	107	1	1	3	-104
8	108	1	1	4	-104
9	109	1	1	5	-104
10	110	1	1	6	-104

We now know enough to revisit `MA_crossover`. Here is the function again.

```
> MA_crossover
```

```

function(m, S) {
  m.fast <- MA(S, m[1], pad = NA)
  m.slow <- MA(S, m[2], pad = NA)

  crossover <- function() {
    if (m.fast[Time()] > m.slow[Time()])
      1
    else
      0
  }
  tail(btest(S, signal = crossover,
            b = 60, initial.cash = 100,
            convert.weights = TRUE)$wealth,
      n = 1)
}

```

The function first computes the two moving averages; within the `signal` function, the current values of these moving averages are then accessed with `Time()`. The backtest is run and its equity curve (`wealth`) is extracted, of which the final value (`tail(..., 1)`) is returned.

## More examples

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

```
> trade_details(btest(prices, signal = function() 5),  
                prices)
```

	price	suggest	position	wealth	cash
1	101	0	0	0	0
2	102	5	5	0	-510
3	103	5	5	5	-510
4	104	5	5	10	-510
5	105	5	5	15	-510
6	106	5	5	20	-510
7	107	5	5	25	-510
8	108	5	5	30	-510
9	109	5	5	35	-510
10	110	5	5	40	-510

An often-used way to specify a trading strategy is to map past prices into +1, 0 or -1 for going long, flat or short. A signal is then often only given at one specified point (as in “buy one unit now”). Example: suppose our rule said “buy after the third period”. To access the current time, we have already seen that can use the function `Time`.

```
> signal <- function()  
  if (Time() == 3L)  
    1 else 0  
> trade_details(btest(prices, signal), prices)
```

	price	suggest	position	wealth	cash
1	101	0	0	0	0
2	102	0	0	0	0
3	103	0	0	0	0
4	104	1	1	0	-104
5	105	0	0	1	1
6	106	0	0	1	1
7	107	0	0	1	1
8	108	0	0	1	1
9	109	0	0	1	1
10	110	0	0	1	1

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

	price	suggest	position	wealth	cash
1	101	0	0	0	0
2	102	0	0	0	0
3	103	0	0	0	0
4	104	1	1	0	-104
5	105	1	1	1	-104
6	106	1	1	2	-104
7	107	1	1	3	-104
8	108	1	1	4	-104
9	109	1	1	5	-104
10	110	1	1	6	-104

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

We may also prefer to specify `signal` so that it evaluates to a weight; for instance, after a portfolio optimization.

```
> signal <- function()
  0.5 ## invest 50% of wealth in asset
```

In such a case, you need to set `convert.weights` to `TRUE`. We also need to have a meaningful initial wealth: 50 percent of nothing is nothing.

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

	price	suggest	position	wealth	cash
1	101	0.000	0.000	100	100.0
2	102	0.495	0.495	100	49.5
3	103	0.490	0.490	100	50.0
4	104	0.488	0.488	101	50.2
5	105	0.486	0.486	101	50.5
6	106	0.483	0.483	102	50.7
7	107	0.481	0.481	102	51.0
8	108	0.479	0.479	103	51.2
9	109	0.476	0.476	103	51.5

```
10    110    0.474    0.474    104    51.7
```

Now something interesting has happened: the function traded in every period from the second. That is because we never invested exactly 50%, which in turn happened because we never knew the price at which we would execute the trade. (Which is the case in practice.) In a backtest we could “cheat,” of course, by looking ahead; and `btest` would allow you to do that (though it will never happen by default).

But a much better way is to check the actual trade sizes:

```
> journal(bt)
```

	instrument	timestamp	amount	price
1	asset 1	2	0.49505	102
2	asset 1	3	-0.00485	103
3	asset 1	4	-0.00236	104
4	asset 1	5	-0.00233	105
5	asset 1	6	-0.00230	106
6	asset 1	7	-0.00227	107
7	asset 1	8	-0.00224	108
8	asset 1	9	-0.00221	109
9	asset 1	10	-0.00218	110

9 transactions

The trades will become very small. So small, in fact, that in practice you would likely not execute them because of transaction costs. To handle such cases, we may either rewrite `signal` so that it checks whether it should trade or not. For many simple (and common) cases, as here, `btest` has an argument `do.rebalance`, which also is a function. It should evaluate to `TRUE` if we want to trade, and `FALSE` otherwise.

```
> dont_if_small <- function() {  
  diff <- SuggestedPortfolio(0) - Portfolio()  
  abs(diff) > 5e-2  
}  
> bt <- btest(prices,  
  signal,  
  convert.weights = TRUE,  
  initial.cash = 100,  
  do.rebalance = dont_if_small)  
> trade_details(bt, prices)
```

	price	suggest	position	wealth	cash
1	101	0.000	0.000	100	100.0
2	102	0.495	0.495	100	49.5
3	103	0.490	0.495	100	49.5
4	104	0.488	0.495	101	49.5
5	105	0.486	0.495	101	49.5
6	106	0.483	0.495	102	49.5
7	107	0.481	0.495	102	49.5
8	108	0.479	0.495	103	49.5
9	109	0.477	0.495	103	49.5
10	110	0.475	0.495	104	49.5

`do.rebalance` is called after `signal`. Hence, the suggested position is known and the lag may be zero (`'SuggestedPortfolio(0)'`).

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

The argument `tol.p` has a slightly different effect: it restricts rebalancing to those trades that lead to position changes of more than  $100 \times \text{tol.p}$  percent.

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

	price	suggest	position	wealth	cash
1	101	0.000	0.000	100	100.0
2	102	0.495	0.495	100	49.5
3	103	0.490	0.495	100	49.5
4	104	0.488	0.495	101	49.5
5	105	0.486	0.495	101	49.5
6	106	0.483	0.495	102	49.5
7	107	0.481	0.495	102	49.5
8	108	0.479	0.495	103	49.5
9	109	0.477	0.495	103	49.5
10	110	0.475	0.495	104	49.5

That basically concludes the tutorial for `btest`. Now let us use a real dataset.

## 3.2 Robert Shiller's Irrational-Exuberance data

Going to a backtest with real data creates a dilemma. On the one hand, we think that computation and its purpose should not be separated—see the discussion in Chapter 1 of Gilli et al.—, and working with real data is much more illuminating and interesting.

On the other hand, the purpose of this chapter is to describe software and techniques; the purpose is not to do empirical analysis. But in the process of backtesting, most of the time must be scheduled first for checking, cleaning and preparing data, and then for analyzing results—crucial tasks, but simply not the topic of this chapter. Still, we decided to include examples based on real datasets. Just keep in mind that the discussions are not meant as complete case studies whose results could be applied as is.<sup>9</sup>

Robert Shiller has collected aggregate data for U.S. stocks for the period 1871 until today, which he used in his research papers and, notably, in his book “Irrational Exuberance” (Shiller, 2000, 2015). The dataset, which he regularly updates, is available from his homepage at <http://www.econ.yale.edu/~shiller/>. The `NMOF` package provides a function `Shiller` that downloads the data and arranges them in a data frame.<sup>10</sup>

```
[shiller-data] > library("NMOF")
> data <- Shiller(dest.dir = "~/Downloads/Shiller")
> str(data)
```

```
'data.frame':  1780 obs. of  10 variables:
 $ Date      : Date, format: "1871-01-31" ...
 $ Price     : num  4.44 4.5 4.61 4.74 4.86 4.82 4..
 $ Dividend  : num  0.26 0.26 0.26 0.26 0.26 0.26 ..
 $ Earnings  : num  0.4 0.4 0.4 0.4 0.4 0.4 0.4 0...
 $ CPI       : num  12.5 12.8 13 12.6 12.3 ...
 $ Long Rate : num  5.32 5.32 5.33 5.33 5.33 ...
 $ Real Price: num  90.3 88.8 89.7 95.7 100.4 ...
 $ Real Dividend: num  5.29 5.13 5.06 5.25 5.37 ...
 $ Real Earnings: num  8.14 7.9 7.78 8.08 8.26 ...
 $ CAPE      : num  NA NA NA NA NA NA NA NA NA NA ..
```

<sup>9</sup>You know, prices may go up or down. Or stay unchanged. (The publisher asked us to write this. :-))

<sup>10</sup>A note for Windows users: the symbol `~` in the function call stands for the user's home directory, which is essentially equivalent to `C:\Users\<username>\`, though this has varied between Windows versions. To use it in R on Windows, the safest way is to set an environment variable `HOME` to the desired path. In R, you may query the value of `~` with `path.expand("~/")`.

We will also use packages `zoo` (Zeileis and Grothendieck, 2005), for handling time-series, and `PMwR`.

```
> library("PMwR")
> library("zoo")
```

[packages]

Shiller's dataset comprises, among other variables, a monthly total-return series of the S&P Composite Index<sup>11</sup> and a time series of aggregate earnings for the stocks included in the index. Taking the ratio of these two series gives us total-market price/earnings, or P/E, ratio, i.e. a measure of the valuation of the stock market as a whole. Shiller averages earnings over a period of ten years (see Shiller, 1996), for which he took inspiration from Graham and Dodd (1934). The result is usually referred to as the Cyclically-Adjusted P/E, or CAPE, ratio. We extract these two series and store them as `zoo` objects `price` and `CAPE`. We also store the dates as a vector `timestamp`.

```
> timestamp <- data$Date
> price <- scale1(zoo(data$Price, timestamp),
                  level = 100)
> CAPE <- zoo(data$CAPE, timestamp)
```

[extract-data]

Figs 11 and 12 provide plots of both series. The code for both graphics is essentially the following.

```
> plot(price,
       xlab = "", ylab = "S&P Composite",
       xaxt = "n", yaxt = "n", log = "y",
       type = "l", lwd = 0)
> abline(h = c(100, axTicks(2)),
        lwd = 0.25, col = grey(0.7))
> axis(2, lwd = 0)
> t <- axis.Date(1, x = data$Date, lwd = 0)
> abline(v = t, lwd = 0.25, col = grey(0.7))
> lines(price, type = "l")
> abline(v = as.Date(c("1929-09-30",
                      "1999-12-31")),
        col = grey(0.5))
```

[shiller1]

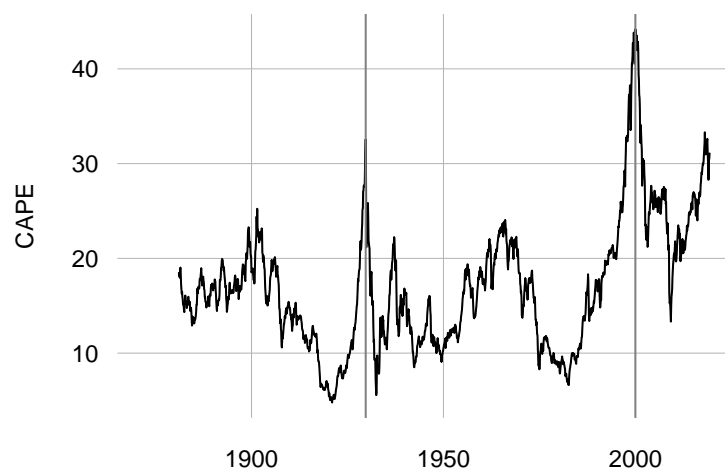
Fig. 12 shows that the aggregate valuation of stocks, when measured as a multiple of realized earnings, fluctuates substantially over time: from below 10

<sup>11</sup>Today, this means the S&P 500. The S&P 500 index dates back to the 1920s, though it was computed from fewer (less than one-hundred) companies back then. Accordingly, it was not called S&P 500, but S&P Composite. Only in 1957 reached the number of stocks in the index 500. See <https://www.britannica.com/topic/SandP-500>.

Figure 11: Time series from Robert Shiller's dataset: S&P stock market index since January 1871. The series starts at 100. The vertical axis uses a log scale. The two vertical lines indicate highest values of the CAPE ratio in the past (Fig. 12): September 1929 and December 1999. Both times were close to market tops.



Figure 12: Time series from Robert Shiller's dataset: Cyclically-adjusted price earnings ratio (CAPE ratio). The two vertical lines indicate highest values in the past: September 1929 and December 1999. Both times were close to market tops.



to more than 40 at its peak in the late 1990s. This fluctuation primarily stems from changes in the price series; the earnings series, because it is smoothed, varies much less. This leads to the idea that the CAPE ratio may give indications as to when the market is overvalued. By this theory, because earnings are rather stable, a high ratio means that prices have overshot their fundamental value, and vice versa. And indeed: after peaks in the CAPE, the markets typically performed poorly: see 1929 and 2000, which we have marked in Figs 11 and 12.

But, as we argued above, such graphical reasoning can easily be deceiving. We could run a straightforward backtest: whenever the CAPE is below 25, we stay fully invested in the index; when above 25, we sell and stay out of the market. But we cannot do that, of course. We could never have known that 25 would be a high level for the CAPE until much later. So we write `signal` differently: we move out of the market when the CAPE exceeds the 90th quantile of its known history.

```
> avoid_high_valuation <- function(CAPE) {
  Q <- quantile(CAPE[n = Time()], 0.9, na.rm = TRUE)
  if (CAPE[Time()] > Q)
    0
  else
    1
}
> bt_avoid_high_valuation <-
  btest(price,
        signal = avoid_high_valuation,
        initial.cash = 100,
        convert.weights = TRUE,
        CAPE = CAPE,
        timestamp = timestamp,
        b = as.Date("1899-12-31"))
```

[avoid-high-valuation]

Remarks: first, in the signal function, named `avoid_high_valuation`, we compute the quantile on the fly. Alternatively, we might have precomputed it for the whole series, as we did in the previous section for the moving average. Second, we specify the burn-in `b` as a date, which is often more convenient than specifying a number. This is only possible, because—third—we specify another argument, `timestamp`, when we call `btest`. Passing the timestamp has a number of benefits. For instance, if you want to see how the strategy fared, it is easiest to extract the equity curve with `as.NAVseries`, which now displays the timestamps.

```
> as.NAVseries(bt_avoid_high_valuation)
```

```
31 Dec 1899 ==> 30 Apr 2019    (1433 data points, 0 NAs)
      100              38219.1
```

For NAVseries, there exists an informative summary method (we suppress some of its output).

```
> summary(as.NAVseries(bt_avoid_high_valuation))
```

```
-----
31 Dec 1899 ==> 30 Apr 2019    (1433 data points, 0 NAs)
      100              38219.1
-----
High              38219.15    (31 Jul 2016)
Low               95.10     (30 Sep 1900)
-----
Return (%)              5.1    (annualised)
-----
Max. drawdown (%)      79.1
_ peak                487.77    (30 Apr 1930)
_ trough              101.89    (30 Jun 1932)
_ recovery                        (31 Aug 1951)
_ underwater now (%)    0.0
-----
Volatility (%)          13.9    (annualised)
_ upside              10.6
_ downside            9.2
-----
```

When timestamp information is available, the actual dates are shown when we extract the trades with `journal`.

```
> journal(bt_avoid_high_valuation)
```

Similarly, we could have passed a character vector `instrument`, giving the actual asset names. If you check the journal, you see that the strategy traded quite often to adjust the actual position. If you rerun the backtest with a restriction on trade size (e.g. by using arguments `tol` or `tol.p`), you will see that these trades did not substantially affect the performance.

Now let us compare the resulting equity curve with the underlying stock market, i.e. a buy-and-hold investment. Because we will do this for other strategies later on, we define two small tools (read: functions). The first function, `merge_series`, takes any number of `zoo`, `btest` or `NAVseries` objects,



and synchronizes them, i.e. it joins them so that their timestamps match. The result is a multivariate zoo series.

```
> merge_series <- function(..., series.names) {
  s <- list(...)
  if (missing(series.names)) {
    if (!is.null(ns <- names(s)))
      series.names <- ns
    else
      series.names <- seq_along(s)
  }
  to_zoo <- function(x) {
    if (inherits(x, "btest")) {
      as.zoo(as.NAVseries(x))
    } else if (inherits(x, "NAVseries")) {
      as.zoo(x)
    } else if (inherits(x, "zoo")) {
      x
    } else
      stop("only zoo, btest and NAVseries are supported")
  }
  s <- lapply(s, to_zoo)
  series <- do.call(merge, s)
  if (is.null(dim(series)))
    series <- as.matrix(series)
  series <- scale1(series, level = 100)
  colnames(series) <- series.names
  series
}
```

[merge-series]

The second function, `series_ratio`, computes the ratio of two series. This is useful to see when one strategy performed better than another (Schumann, 2013).

```
> series_ratio <- function(t1, t2) {
  if (missing(t2))
    scale1(t1[, 1]/t1[, 2], level = 100)
  else
    scale1(t1/t2, level = 100)
}
```

[series-ratio]

So let us use these tools. We first merge the backtest with the S&P data.

```
> series <- merge_series(
```

[series]

Figure 13: Results of avoiding high valuation. The S&P index is shown in black; the strategy in gray.



```
"btest" = bt_avoid_high_valuation ,
"S&P" = price)
```

From these series, we create two plots, shown in Figs 13 and 14. Fig. 14 shows the ratio of the strategy's equity curve and the benchmark. When the numerator (i.e. the strategy) performs better than the denominator (i.e. the benchmark), the line rises; when both series grow at the same rate, the line is flat; and when the benchmark performs better than the strategy, the line declines. For a long time the line is flat, which happens when the strategy is invested in the market. In the 1929 crash the strategy actually did well, though not at once: it first underperformed by about 20%, as the strategy went out of the market and the market kept on rising. When the stock prices plummeted, however, the strategy was vindicated and left with an outperformance of more than 30%. That is, an investor in the strategy would have had a 30% higher terminal wealth than an investor in the benchmark. In the 1990s, however, the strategy often was not invested, yet the market failed to crash. Hence, the line declines.

These results are somewhat sobering. The original figures (11 and 12) had indicated that high valuations were a sign of market tops, but our strategy created from the valuation measure failed to profit substantially, despite substantial declines in the market in the years that followed the tops. The strategy got out of the market, but then went back in too quickly, notably in the 1930s; and it clearly missed out on large parts of the bull markets since the 1990s.

Let us see how our choice of the 90th quantile affects the results. Before you object: the goal is not to find some optimal level for the quantile, but to get an impression of the influence the choice has. To do this, we first make the quantile an argument to the `signal` function.

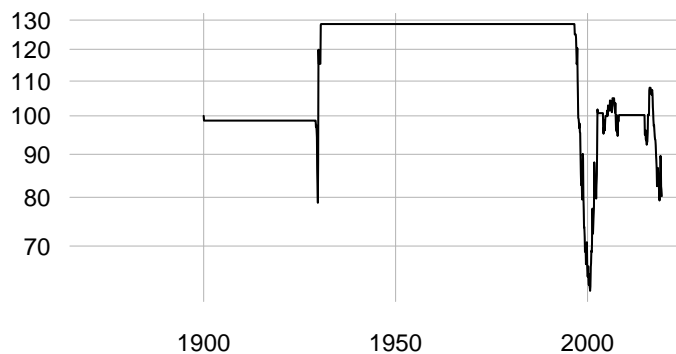


Figure 14: Results of avoiding high valuation compared with S&P 500.

```
> avoid_high_valuation_q <- function(CAPE, q) {
  Q <- quantile(CAPE[n = Time()], q, na.rm = TRUE)
  if (CAPE[Time()] > Q)
    0
  else
    1
}
```

[avoid-high-valuation-q]

Then we fix a vector of values for `q`.

```
> q.values <- c(0.6, 0.7, 0.8,
  seq(0.90, 0.99, by = 0.01))
```

The straightforward way would be now to loop over these values, each time call `btest` and store the results we are interested in. The order in which we run these backtests is not relevant, as the tests are independent from one another. This suggests that we may distribute the computation. There are several ways in which we may do this, and we outline a simple one.<sup>12</sup> First, for each variation of the backtest that we want to run, we collect all arguments to the call to `btest` in a list `args`.

```
> args <- vector("list", length(q.values))
> names(args) <- as.character(q.values)
> for (q in q.values)
  args[[as.character(q)] <-
    list(coreddata(price),
```

[collect-args]

<sup>12</sup>A downside of this approach is that we may make copies of all the data, and move these data to the node at every call. An appendix to this chapter outlines alternative strategies.

```

    signal = avoid_high_valuation_q,
    initial.cash = 100,
    convert.weights = TRUE,
    CAPE = CAPE,
    q = q,
    timestamp = timestamp,
    b = as.Date("1899-12-31"))

```

It is easy to memorize this pattern of setting up the parallel computation because it resembles calling `btest` in a loop: only instead of calling `btest`, we pack all arguments that we would have used into a list.

```

[compare-methods] > ## serial
> variations_q_serial <- vector("list", length(q.values))
> names(variations_q_serial) <- as.character(q.values)
> for (q in q.values)
  variations_q_serial[[as.character(q)]] <-
    btest(coredata(price),
          signal = avoid_high_valuation_q,
          initial.cash = 100,
          convert.weights = TRUE,
          CAPE = CAPE,
          q = q,
          timestamp = timestamp,
          b = as.Date("1899-12-31"))

> ## parallel
> library("parallel")
> ## parallel: socket cluster
> cl <- makePSOCKcluster(rep("localhost", 2))
> ignore <- clusterEvalQ(cl, require("PMwR"))
> variations_q_parallel1 <-
  clusterApplyLB(cl, args,
                 function(x) do.call(btest, x))
> names(variations_q_parallel1) <- as.character(q.values)
> stopCluster(cl)
> ## parallel: fork cluster
> cl <- makeForkCluster(nnodes = 4)
> ignore <- clusterEvalQ(cl, require("PMwR"))
> variations_q_parallel2 <- clusterApplyLB(cl, args,
                                           function(x) do.call(
                                             btest, x))
> names(variations_q_parallel2) <- as.character(q.values)
> stopCluster(cl)

```

Compare the results.

```
> all.equal(variations_q_serial, variations_q_parallel1)
```

```
[1] TRUE
```

```
> all.equal(variations_q_serial, variations_q_parallel2)
```

```
[1] TRUE
```

Such computations are useful so often that `btest` already implements this distribution of computations. So instead of the setup we described above, we could have used the following code (which uses a fork cluster; on Windows, say `method = "parallel"` instead).

```
> variations_q <-  
  btest(coredata(price),  
        signal = avoid_high_valuation_q,  
        initial.cash = 100,  
        convert.weights = TRUE,  
        CAPE = CAPE,  
        timestamp = timestamp,  
        b = as.Date("1899-12-31"),  
        variations = list(q = q.values),  
        variations.settings =  
          list(method = "multicore",  
               cores = 4,  
               label = as.character(q.values)))  
> all.equal(variations_q, variations_q_serial)
```

[variations-q]

```
[1] TRUE
```

Whether we use a loop or not, we end up with a list of backtests. We extract the equity curves of these backtests and plot them. The results are shown in in Fig. 15.

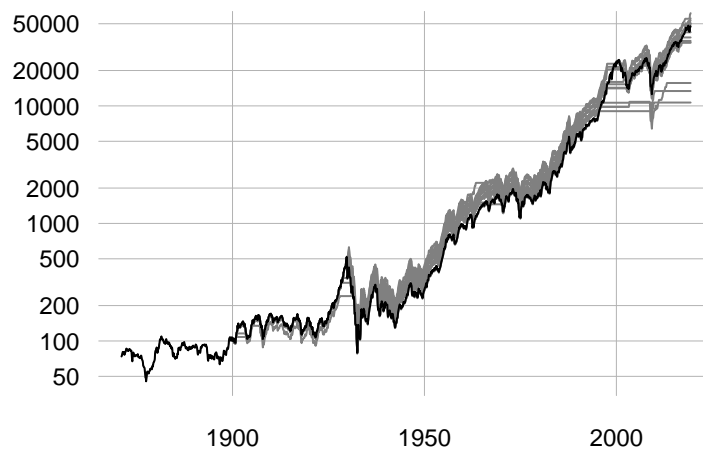
```
> series_var <- do.call(merge_series, variations_q)
```

[series-var]

(Such a one-line piece of code should make clear why it was useful to prepare tools such as `merge_series`.)

In Fig. 15 all gray lines are above the black market line, which shows that for all values of `q`, we would have done somewhat better (at least not worse) than the market in the 1929 crash. However, this outperformance is eroded and in some cases lost in the 1990s. If we look at the whole series, we find

Figure 15: Performance of strategy for different values of  $q$ . The S&P index is shown in black; the strategy variations in gray.



that leaving the market when valuations are truly extreme led to good results. In this way the strategy avoided parts of the two crashes, while at the same time it stayed long enough in the market before the markets dropped. We can see this most easily by comparing final wealth, i.e. total returns.

[returns-total]

```
> returns(window(series[, "S&P"],
               start = as.Date("1901-01-31"),
               end = as.Date("2017-12-31")),
           period = "itd") ## inception to date
```

```
37585.1% [31 Jan 1901 -- 31 Dec 2017]
```

```
> returns(window(series_var,
               start = as.Date("1901-01-31"),
               end = as.Date("2017-12-31")),
           period = "itd")
```

```
0.6: 9775.7% [31 Jan 1901 -- 31 Dec 2017]
0.7: 11443.1% [31 Jan 1901 -- 31 Dec 2017]
0.8: 13411.3% [31 Jan 1901 -- 31 Dec 2017]
0.9: 32867.8% [31 Jan 1901 -- 31 Dec 2017]
0.91: 29990.0% [31 Jan 1901 -- 31 Dec 2017]
0.92: 30711.9% [31 Jan 1901 -- 31 Dec 2017]
0.93: 29600.5% [31 Jan 1901 -- 31 Dec 2017]
0.94: 36361.2% [31 Jan 1901 -- 31 Dec 2017]
0.95: 42231.2% [31 Jan 1901 -- 31 Dec 2017]
```

```
0.96: 47440.0% [31 Jan 1901 -- 31 Dec 2017]
0.97: 41583.1% [31 Jan 1901 -- 31 Dec 2017]
0.98: 40555.6% [31 Jan 1901 -- 31 Dec 2017]
0.99: 39991.6% [31 Jan 1901 -- 31 Dec 2017]
```

Waiting for extreme valuation level was necessary in the 1990s: stay in the market for as long as you could. But this behavior was less advantageous in the 1930s.

```
> returns(window(series[, "S&P"],
                start = as.Date("1920-12-31"),
                end = as.Date("1950-12-31")),
            period = "itd")
```

```
190.0% [31 Dec 1920 -- 31 Dec 1950]
```


```
> returns(window(series_var,
                start = as.Date("1920-12-31"),
                end = as.Date("1950-12-31")),
            period = "itd")
```

```
0.6: 380.1% [31 Dec 1920 -- 31 Dec 1950]
0.7: 381.7% [31 Dec 1920 -- 31 Dec 1950]
0.8: 231.9% [31 Dec 1920 -- 31 Dec 1950]
0.9: 277.8% [31 Dec 1920 -- 31 Dec 1950]
0.91: 238.8% [31 Dec 1920 -- 31 Dec 1950]
0.92: 238.8% [31 Dec 1920 -- 31 Dec 1950]
0.93: 238.8% [31 Dec 1920 -- 31 Dec 1950]
0.94: 249.4% [31 Dec 1920 -- 31 Dec 1950]
0.95: 319.9% [31 Dec 1920 -- 31 Dec 1950]
0.96: 336.7% [31 Dec 1920 -- 31 Dec 1950]
0.97: 290.5% [31 Dec 1920 -- 31 Dec 1950]
0.98: 190.0% [31 Dec 1920 -- 31 Dec 1950]
0.99: 190.0% [31 Dec 1920 -- 31 Dec 1950]
```

You see that now the finding is somewhat reversed: waiting until the last moment was not the most beneficial strategy in the 1920s.

We will leave it at that. The moral: first, it is not easy to move from a potentially-valuable indicator, such as stock-market valuation, to a profitable investment strategy. Second, as we recommended above, one should try to understand why a strategy works. In the case of Robert Shiller's data, the strategy gained its advantage from moving out of the market at critical times; these results are not refutable. But the performance gain comes from only

two episodes that occurred over a period of 150 years, which is not a lot of observations. As we said above, analyzing such results and drawing conclusions from it is not easy: leaving the market too early can, in relative terms, be just as costly as staying in when the market crashes. Alan Greenspan made his famous irrational exuberance remarks in 1996 (after a testimony by Robert Shiller and John Campbell).

When we look at the S&P (with dividends included) between January 1996 and today  the index never saw its 1996-level again, not even in 2000–02 or 2007–08.

Finally, we should stress that the example does not imply that the CAPE ratio “does not work” as an indicator of when to invest (which is something that cannot be shown in any case). But the example demonstrates that a picture may be much more promising than the actual implementation.

## 4 Backtesting portfolio strategies

(That is, we look at multi-asset backtests.)

### 4.1 Kenneth French’s data library

Kenneth French collects, updates, and makes available a treasure trove of datasets of U.S. American equity markets on his website at <http://mba.tuck.dartmouth.edu/pages/faculty/ken.french/>. Many of these data are derived from the CRSP database (<http://www.crsp.com/>). In this section, we work with a dataset comprising industry returns, as described in Fama and French (1997).

We first load the data. The NMOF package provides a convenience function French for downloading and processing the data. As before, we will make much use of the zoo package for handling time-series.

```
[french-data] > library("NMOF")
> library("PMwR")
> library("zoo")
> P <- French("~/Downloads/French",
               dataset = "49_Industry_Portfolios_daily_CSV.zip",
               weighting = "value",
               frequency = "daily",
               price.series = TRUE,
               na.rm = TRUE)
```

We will use only a subset of the series, starting in 1990. We also drop the 49th industry (“Other”). The subset was only chosen to make the analysis more



tractable; you may easily adjust the code to work with the full history (or some other dataset). The return series are transformed to a price series and stored as a zoo series P. We store the timestamp separately. Table 1 presents summary statistics of the single series.

```
> START <- as.Date("1990-01-01")
> END <- as.Date("2018-07-31")

> ## make zoo series
> P <- zoo(P, as.Date(row.names(P)))
> P <- window(P, start = START, end = END)
> timestamp <- index(P)
> P <- P[, colnames(P) != "Other"]

> short <- colnames(P)

> instrument <- colnames(P)
> defs <- French("~/Downloads/French", "Siccodes49.zip")
> data.frame(Abbr = colnames(P),
             Description = defs[match(colnames(P), defs$abbr), "
             industry"])
```

[prepare-data]

Abbr	Description
1 Agric	Agriculture
2 Food	Food Products
3 Soda	Candy & Soda
4 Beer	Beer & Liquor
5 Smoke	Tobacco Products
6 Toys	Recreation
7 Fun	Entertainment
8 Books	Printing and Publishing
9 Hshld	Consumer Goods
10 Clths	Apparel
11 Hlth	Healthcare
12 MedEq	Medical Equipment
13 Drugs	Pharmaceutical Products
14 Chems	Chemicals
15 Rubbr	Rubber and Plastic Products
16 Txtls	Textiles
17 BldMt	Construction Materials
18 Cnstr	Construction
19 Steel	Steel Works Etc
20 FabPr	Fabricated Products

21	Mach	Machinery
22	ElcEq	Electrical Equipment
23	Autos	Automobiles and Trucks
24	Aero	Aircraft
25	Ships	Shipbuilding, Railroad Equipment
26	Guns	Defense
27	Gold	Precious Metals
28	Mines	Non-Metallic and Industrial Metal Mining
29	Coal	Coal
30	Oil	Petroleum and Natural Gas
31	Util	Utilities
32	Telcm	Communication
33	PerSv	Personal Services
34	BusSv	Business Services
35	Hardw	Computers
36	Softw	Computer Software
37	Chips	Electronic Equipment
38	LabEq	Measuring and Control Equipment
39	Paper	Business Supplies
40	Boxes	Shipping Containers
41	Trans	Transportation
42	Whlsl	Wholesale
43	Rtail	Retail
44	Meals	Restaurants, Hotels, Motels
45	Banks	Banking
46	Insur	Insurance
47	RlEst	Real Estate
48	Fin	Trading

We start by looking at the data (a.k.a. plotting). The results are shown in Fig. 16.

```
[industries-series] > plot(scale1(P, level = 100),
      plot.type = "single",
      log = "y",
      col = grey.colors(ncol(P)),
      xlab = "",
      ylab = "")
```

There clearly was an uptrend in prices; and there also was substantial variation in the cross-section of industry returns. But apart from that, it is difficult to read the chart, as the different lines can hardly be discerned from one another. A useful alternative way to plot such a collection of series is a fanplot. The following code block shows how it may be constructed.

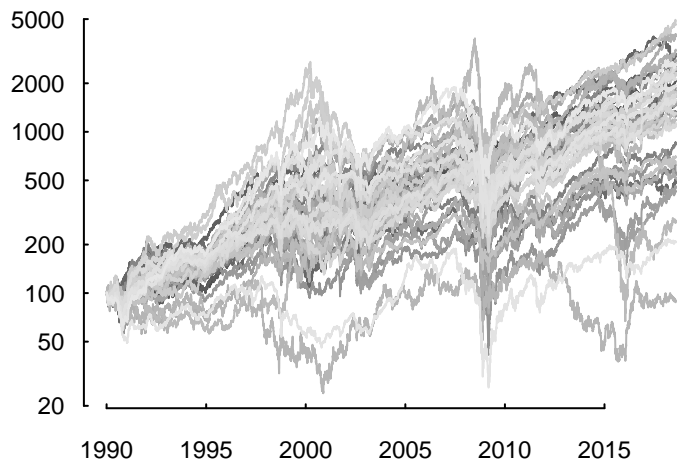


Figure 16: Performance of industry portfolios as provided by Kenneth French's data library, January 1990 to May 2018. Time series are computed from daily returns.

```
> P100 <- scale1(P, level = 100) ## P must be 'zoo'
> P100 <- aggregate(P100,
                     datetimetools::end_of_month(index(P100)),
                     tail, 1)
> nt <- nrow(P100)
> levels <- seq(0.01, 0.49, length.out = 10)
> greys <- seq(0.9, 0.50, length.out = length(levels))

> ### start with an empty plot ...
> plot(index(P100), rep(100, nt), ylim = range(P100),
       xlab = "", ylab = "",
       lty = 0,
       type = "l",
       log = "y")

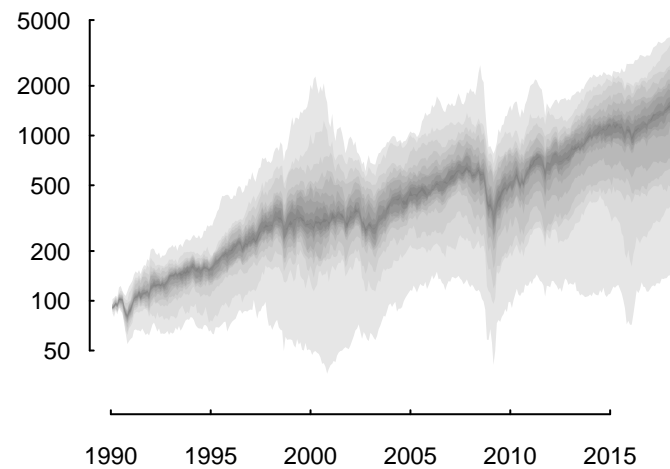
> ### ... and add polygons
> for (level in levels) {

  l <- apply(P100, 1, quantile, level)
  u <- apply(P100, 1, quantile, 1 - level)
  col <- grey(greys[level == levels])
  polygon(c(index(P100), rev(index(P100))),
         c(l, rev(u)),
         col = col,
         border = NA)

}
```

[fanplot]

Figure 17: Fanplot of the industry time-series shown in Fig. 16.



The results, shown in Fig. 17, look much clearer. And as you will see later, when we add other benchmarks, the implied trend is very reasonable. Of course, such code should not be run as a script, so we move the code into a function `fan_plot`, which we will reuse later.

Besides the total returns of the different industries, we should also be interested in how they move together. For the sake of not making an already long chapter even longer, we limit ourselves to computing pairwise correlations. But analyzing co-movement between the assets is an important question, and there are different ways to answer it. In fact, it would be best to receive and compare several answers, e.g. from factor analysis, or PCA.

```
[correlations] > C <- cor(returns(P, period = "month"))
> cors <- C[lower.tri(C)]
> summary(cors)
```

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
0.012	0.379	0.503	0.488	0.616	0.854

A histogram of these correlations is shown in Fig. 18.

Now that we have seen the data, let us create benchmarks. First, also from Kenneth French's data library, we take a total-market index. We store it as a zoo series named `market`.

```
[market] > ### ... market
> ff3 <- French("~/Downloads/French",
```

Table 1: Statistics of the industry series. Returns are annualised, in percent. Volatility is computed from monthly returns and also annualised.






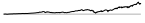

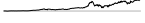












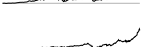










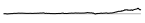
















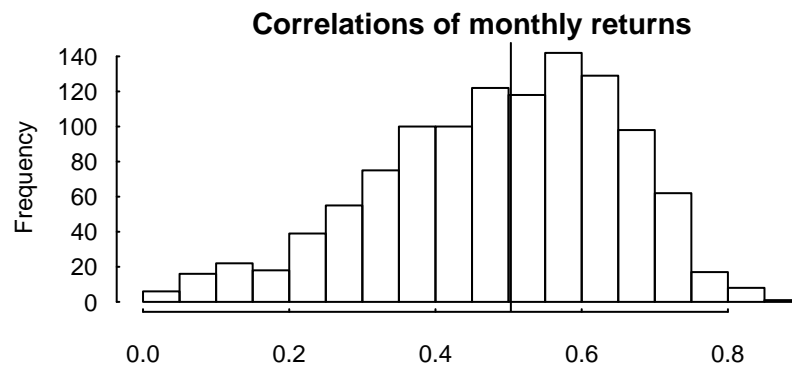
Short name	Return in %	Volatility in %		Short name	Return in %	Volatility in %	
Softw	14.6	25.3		BusSv	10.4	17.0	
Guns	14.1	20.5		Food	10.4	13.8	
Aero	14.1	20.4		BldMt	10.3	21.3	
Ships	13.6	25.5		Agric	10.5	21.1	
MedEq	13.1	16.6		Hshld	9.9	14.4	
Smoke	13.1	23.0		Boxes	9.7	20.5	
ElcEq	12.7	21.4		Oil	9.6	18.6	
Fin	12.4	23.5		Cnstr	9.4	23.7	
Beer	12.4	16.4		Util	9.3	13.6	
Rtail	12.3	17.0		Paper	9.3	17.3	
Chips	12.0	27.5		Mines	9.0	26.9	
Fun	12.3	26.4		Whsl	9.1	15.7	
Drugs	12.1	15.9		Hlth	8.5	21.7	
LabEq	12.1	22.5		PerSv	7.6	20.0	
Meals	11.8	16.4		Telcm	7.4	17.4	
Soda	11.7	24.0		Txtls	6.6	27.3	
Insur	11.5	18.0		Books	6.6	18.9	
Rubbr	11.3	19.8		Autos	6.2	25.9	
Chems	10.9	19.1		Toys	6.3	21.7	
Trans	10.9	17.9		FabPr	5.7	26.1	
Hardw	10.6	27.5		Steel	5.6	28.3	
Clths	10.8	21.3		Coal	3.8	42.1	
Banks	10.6	21.2		REst	3.0	25.9	
Mach	10.3	22.6		Gold	-0.7	37.9	

Figure 18: Correlations of monthly returns.



```

      "F-F_Research_Data_Factors_daily_CSV.zip",
      frequency = "daily",
      price.series = TRUE,
      na.rm = TRUE)
> save(ff3, file = "~/Desktop/ff3.RData")
> load(file = "~/Desktop/ff3.RData")
> market <- zoo(ff3[["Mkt-RF"]]*ff3[["RF"]],
               as.Date(row.names(ff3)))
> market <- scale1(window(market, start = START, end = END),
                    level = 100)

```

A second useful benchmark is an equally-weighted portfolio. (We shall from now refer to this strategy as EW.) Computing it will be our first multivariate use of `btest`.

### An equally weighted portfolio

In the previous section we saw that we can access prices in the signal function with `Close`. For more than one asset, `btest` works just the same, only `Close()` will now give us prices for all assets. More specifically, it will return one row of the prices matrix. So, for an equally-weighted portfolio, the signal function is straightforward.

```

[ew] > ew <- function() {
      n <- ncol(Close())
      rep(1/n, n)
    }

```

Note that we could have saved a little code (and computing time) by making the number of assets `n` a constant to be passed, or by precomputing `w`. But running the backtest takes less than a second, so this seems unnecessary.

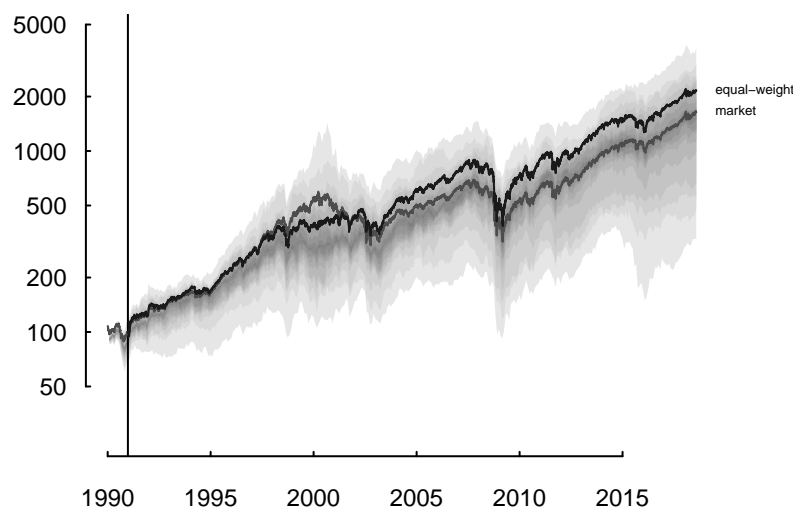


Figure 19: Benchmarks: Performance of market portfolio and of equally-weighted portfolio. The gray shades are the same as in Fig. 17 and indicate the range of performance of the different industries.

```
> bt.ew <- btest(prices = list(coredata(P)),
  signal = ew,
  do.signal = "lastofmonth",
  convert.weights = TRUE,
  initial.cash = 100,
  b = 250,
  timestamp = timestamp,
  instrument = instrument)
```

[bt-ew]

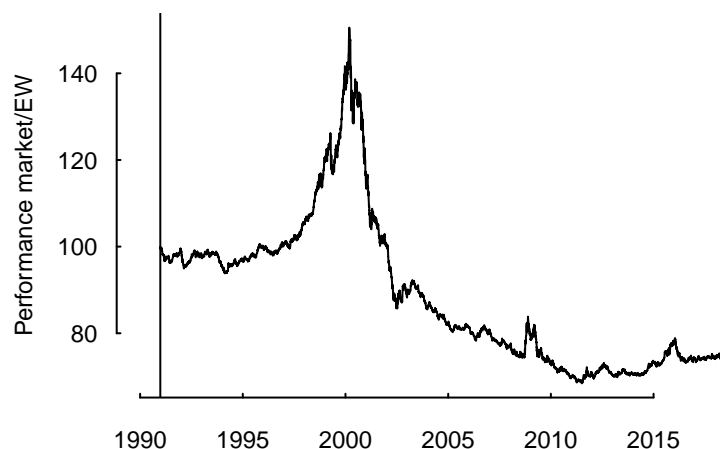
Note the value to `do.signal`: `btest` will extract the last trading day of each quarter in `timestamp` and compute and rebalance on those dates.

Fig. 19 shows the market series and also the performance of the equally-weighted portfolio.

## 4.2 Momentum

The abnormally-high “returns of buying winners and selling losers”, a.k.a. momentum returns, were described in Jegadeesh and Titman (1993) and have proved to be perhaps the most-widely replicated anomaly. Moskowitz and Grinblatt (1999) report evidence that cross-sectional industry momentum could explain much of the previously described single-stock momentum. In this section, we are going to test such a strategy on the industry portfolios obtained from Kenneth French’s Data Library. The purpose of this section is to demonstrate the functionality of the `btest` function in a multi-asset

Figure 20: Benchmarks: Out-performance of market versus equally-weighted portfolio.



backtest, and also show how backtests may be analyzed and evaluated.

### The setup

Suppose we wish to invest in the 10 sectors that have had the highest returns over the past year. We want to give equal weight to each industry. We have already collected the total return series in a matrix  $P$ . The momentum computation for a given point in time is conveniently put into a function `mom`:

```
[mom] > mom <- function(P, k) {
      o <- order(P[nrow(P), ], P[1, ], decreasing = TRUE)
      w <- numeric(ncol(P))
      w[o[1:k]] <- 1/k
      w
    }
```

The computation assumes that the first row of  $P$  contains the price one year ago; and the last row contains the most recent price. The function does not care whether we have data of daily or monthly or any other frequency, nor does it care about the number of rows in  $P$ .

We can best demonstrate how `mom` works through a small example. Suppose we have just three assets, with returns 10%, 20%, and 30%. We wish to be equally weighted in the best two assets, i.e.  $k$  is 2.<sup>13</sup>

---

<sup>13</sup>The example illustrates an important advantage of collecting all computations in a function: we can easily write test cases for the function—an activity that is crucial in practical applications.



```
> P3 <- matrix(c( 1 , 1 , 1,
                  1.1, 1.2, 1.3),
               nrow = 2, byrow = TRUE)
> mom(P3, k = 2)
```

[mom-test]

```
[1] 0.0 0.5 0.5
```

Suppose we had wanted to invest only in the asset with the highest return, i.e.  $k$  is 1.

```
> mom(P3, k = 1)
```

```
[1] 0 0 1
```

Let us test `mom` with more realistic data: we use the first 250 rows of the dataset. Note that in the following code block, it is necessary to say `coredata(P)`. See Appendix A as to why.

```
> mom.latest <- mom(head(coredata(P), 250), 10)
> table(mom.latest)
```

```
mom.latest
 0 0.1
38 10
```

```
> df <- data.frame(sector = instrument,
                   weight = mom.latest)
> df[df$weight > 0, ]
```

	sector	weight
2	Food	0.1
4	Beer	0.1
5	Smoke	0.1
12	MedEq	0.1
13	Drugs	0.1
26	Guns	0.1
31	Util	0.1
35	Hardw	0.1
36	Softw	0.1
40	Boxes	0.1

The function `mom` itself follows a simple pattern: take as input the data matrix (prices, sorted in time) and optionally other parameters; return the

target portfolio. Now we need to make `btest` understand `mom`. Recall that `btest` takes an argument `signal`: a function that returns the target position. A generic implementation of `signal` takes a function and its parameters as inputs.

```
[signal-fun] > signal <- function(fun, ...) {
               P <- Close(n = 250)
               fun(P, ...)
             }
```

You may wonder, why such a nested structure? The reason is code reuse: `mom` is a function that is in no way related on `btest`, and might be used separately. `signal`, in turn, is responsible for setting up the data `P` and then calling `fun`; but it does not need to know what `fun` actually does. With `signal` defined, we may run `btest`.

```
[bt-mom] > bt.mom <- btest(prices = list(P),
                        signal = signal,
                        do.signal = "lastofmonth",
                        convert.weights = TRUE,
                        initial.cash = 100,
                        k = 10,
                        fun = mom,
                        b = 250,
                        timestamp = timestamp,
                        instrument = instrument)
```

The equity curve of `bt.mom` is shown in Fig. 21. We can summarize the backtest results with `summary`.

```
> summary(as.NAVseries(bt.mom), na.rm = TRUE)
```

```
-----
26 Dec 1990 ==> 31 Jul 2018   (6953 data points, 0 NAs)
      100             6345.63
-----
High             6487.71   (26 Jan 2018)
Low              93.57    (09 Jan 1991)
-----
Return (%)             16.2   (annualised)
-----
Max. drawdown (%)      52.8
_ peak                2952.01   (23 Jun 2008)
_ trough              1394.19   (09 Mar 2009)
```

_ recovery	(17 Jan 2013)
_ underwater now (%)	2.2
-----	
Volatility (%)	15.9 (annualised)
_ upside	13.5
_ downside	9.6
-----	

Overall, there seems little doubt that the momentum portfolio performed well. But it should also be of interest when it performed well. We first merge the backtest results with the benchmarks.

```
> series.mom <- merge_series(
  Momentum = bt.mom,
  "Equal-weight" = bt.ew,
  Market = market)
```

[series.mom]

Now we may use the `series_ratio` to plot the ratio of the equity curves, in order to visualize the relative performance. The result is shown in Fig. 22. The graphic shows the ratio of the strategy's equity curve and the benchmark. When the numerator (i.e. the strategy) performs better than the denominator (i.e. the benchmark), the line rises; when both series grow at the same rate, the line is flat; and when the benchmark performs better than the strategy, the line declines.

Fig. 22 illustrates that the strategy performed well in 2000–03, and also in the run-up to the financial crisis, during which it even underperformed the market. Since then, during the “new normal” boom, it has performed roughly in line with the market.

```
> plot(series.mom,
  plot.type = "single", log = "y",
  col = c(grey(0), col.market, col.ew),
  ylab = paste("Growth of USD 100 since",
    as.character(attr(series.mom, "scale1_origin"))))
> abline(v = attr(series.mom, "scale1_origin"))
```

[momentum1]

```
> plot(series.mom[, "Market"], log = "y",
  col = grey(.5), ylab = "Performance", xlab = "")
> ## lines(series.mom[, "Momentum"])
> lines(series_ratio(
  series.mom[, c("Momentum", "Market")]),
  ylab = "Performance Momentum/Market", xlab = "")
```

[momentum2]

Figure 21: Performance of momentum strategy. The vertical axis uses a log scale.

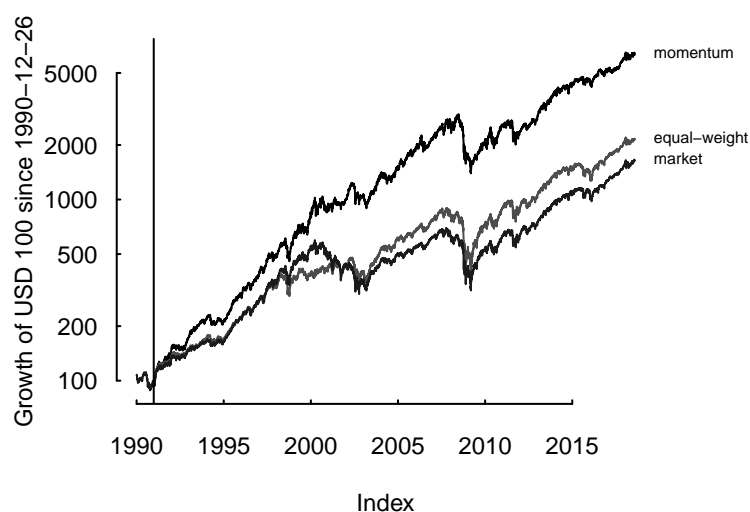
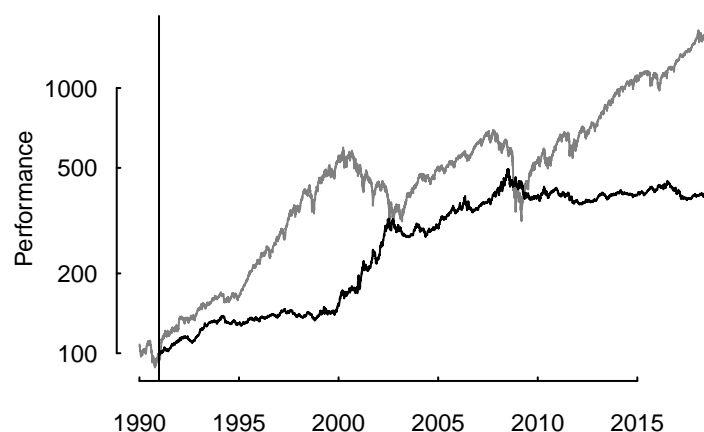


Figure 22: Performance of momentum strategy. The vertical axis uses a log scale. The gray line shows the absolute performance of the market. The black line shows the relative performance of momentum compared with the market, i.e. a rising line indicates an outperformance of momentum.



```
> abline(v = attr(series.mom, "scale1_origin"))
```

## Sensitivity

We emphasized before that any backtest depends on many settings. Variables for momentum are:

- The data: is the performance driven by only a few assets, or only by specific time periods?
- The size of the portfolio ( $k$ ), and also its weighting scheme.
- The choice of rebalancing period: do we have to switch to the target portfolio quickly, or does rebalancing not matter much?

- The lookback period.

We are not going to go through each variable, but limit ourselves to principles and some examples. Computationally, testing variations is straightforward. Suppose we wished to investigate the influence of the lookback period. We would rewrite `signal` to take a parameter `hist`. We could then fix interesting values for `hist` and run a loop.

We may also want to check the interaction between different inputs, which would require nested loops: one loop for each variable of interest. Since the number combinations explodes quickly in such cases, we can often only (randomly) sample from these combinations.

In any case, the result of such an analysis will be a collection of backtests, i.e. a collection of out-of-sample paths. So for any computation we would have done pathwise, we now have a distribution; e.g., a distribution of returns or volatilities. The first task then is to explore these distributions. A robust strategy should not show a breakdown of performance along some paths, and also not very variable performance. Very interesting is also to compare families of settings. For instance, for momentum, one family may comprise backtests that capture short-term momentum (i.e. often rebalanced), and the other family long-term momentum.

But let us return to the process of computing the backtests. Even if we look only at a single input, it may take a lot of time to compute results. We may reduce the wallclock time by distributing the computations.

Let us run a concrete example: let us see the effect of the rebalancing frequency on the returns of the momentum strategy. The following code chunk simulates 200 rebalancing schemes: the 100 rebalance frequently, with 5 to 25 days between rebalancing. The second 100 schemes rebalance less frequently, with 26 to 300 days between rebalancing. Note that neither scheme includes transaction fees.

```
> library("parallel")
> runs <- 100
> args_short <- vector("list", length = runs)
> args_long <- vector("list", length = runs)
> for (i in seq_len(runs)) {
  when <- cumsum(c(250, sample(5:25, 5000, replace=TRUE)))
  when <- when[when <= length(timestamp)]

  args_short[[i]] <- list(prices = list(coredata(P)),
                        signal = signal,
                        do.signal = when,
                        convert.weights = TRUE,
```

[long-short]

```

        initial.cash = 100,
        k = 10,
        fun = mom,
        b = 550,
        ## tc = 0.0025,
        timestamp = timestamp,
        instrument = instrument)

when <- cumsum(c(250, sample(26:300, 5000, replace=TRUE)))
when <- when[when <= length(timestamp)]

args_long[[i]] <- list(prices = list(coredata(P)),
                      signal = signal,
                      do.signal = when,
                      convert.weights = TRUE,
                      initial.cash = 100,
                      k = 10,
                      fun = mom,
                      b = 550,
                      ## tc = 0.0025,
                      timestamp = timestamp,
                      instrument = instrument)

}
> cl <- makePSOCKcluster(rep("localhost", 4))
> clusterEvalQ(cl, require("PMwR"))

```

```

[[1]]
[1] TRUE

[[2]]
[1] TRUE

[[3]]
[1] TRUE

[[4]]
[1] TRUE

```

```

> variations1 <- clusterApplyLB(cl, args_short,
                                function(x) do.call(btest, x))
> variations2 <- clusterApplyLB(cl, args_long,
                                function(x) do.call(btest, x))

```

```

> stopCluster(cl)
> mom_vars1 <- do.call(merge_series, variations1)
> mom_vars2 <- do.call(merge_series, variations2)

```

Results are summarized in Figs 23 to 25. We get a clear picture: rebalancing often performs on average better than waiting longer. But note that this is not the final verdict. After all, we did not consider transaction costs (though at the level of industries, as we look at the strategies, they will not matter too much, since they are cap-weighted).

### 4.3 Portfolio optimization

This section has three goals: i) show how a portfolio model, such as one of those discussed in Gilli et al., Chapter 14, may be backtested with `btest`; ii) show the sensitivity of such a model to the specific settings under which it is run, and iii) show the effects of using a non-exact method instead of an exact one.

We will discuss a relatively simple model: computing a low-risk portfolio, with risk being the variance. The advantage of this model is that it can be solved exactly via quadratic programming. Thus, we may compare the exact solution with the one obtained by a heuristic.

#### The setup

Within the `signal` function there are now two things to do: first, receive historical data and compute a forecast of the variance–covariance matrix. We collect these computations in a function `cov_fun`. Second, given a variance–covariance matrix, and perhaps other restrictions, compute the MV weights. These computations will go into function `mv_fun`. We may thus write down prototypes for both functions:

```

> cov_fun <- function(R, ...) {
  ## ....
}
> mv_fun <- function(var, wmin, wmax) {
  ## ....
}

```

[prototypes]

The reason for separating these computations is that we may want to try different methods for both computations. The actual signal function is provided in `signal_mv`. As before, its job is to fetch the data via `Close()`, and then call functions `cov_fun` and `mv_fun`. One point to mention is the parameter `n`: we pick prices every `n` days and compute returns. Below, we set

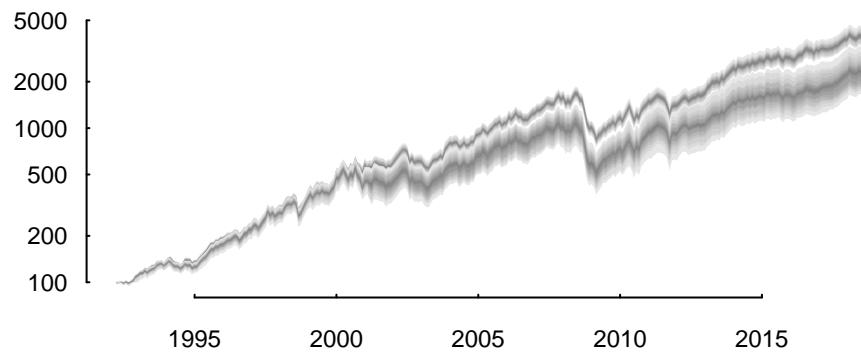


Figure 23: Results from sensitivity check: Fanplots of paths with frequent rebalancing (the upper band) and less frequent rebalancing (the lower band).

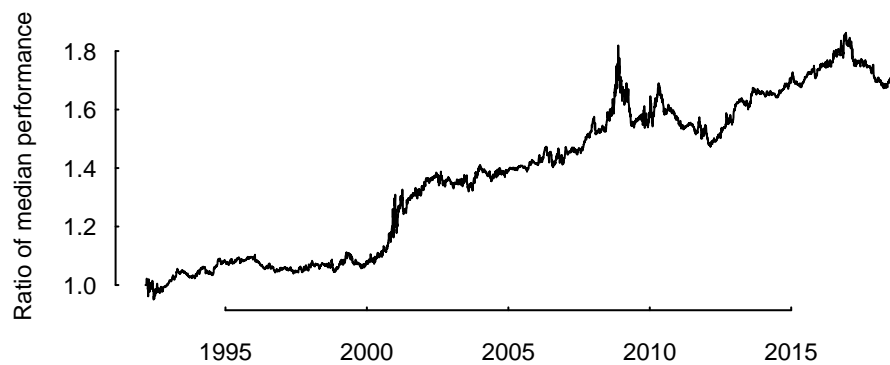


Figure 24: Results from sensitivity check: The ratio of the median paths of the fanplots in Fig. 23. The steadily rising line indicates that frequent rebalancing outperforms less frequent rebalancing.

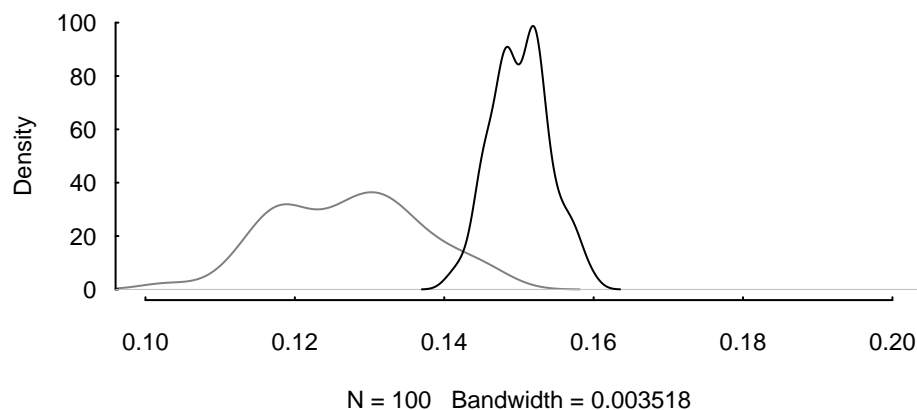


Figure 25: Results from sensitivity checks: densities of annualised returns of the fanplots in Fig. 23. The distribution to the right belongs to the paths with more frequent rebalancing.



n to 20. In this way, we at least mitigate spurious “uncorrelatedness”, which results from even small asynchronicities in the industry series.<sup>14</sup>

```
> signal_mv <- function(cov_fun, mv_fun,
                        wmin, wmax, n, ...) {
    ## cov_fun .. takes a matrix R of returns
    ##               (plus ...), and evaluates to
    ##               the variance--covariance matrix
    ##               of those returns
    ##
    ## mv_fun .. takes a covariance matrix and
    ##               min/max weights, and returns
    ##               minimum-variance weights

    P <- Close(n = 2500)
    i <- seq(1, nrow(P), by = n)
    R <- PMwR::returns(P[i, ])
    cv <- cov_fun(R, ...)
    mv_fun(cv, wmin, wmax)
}
```

[signal-mv]

The default `mv_fun` uses the function QP solver from the `quadprog` package. Accordingly, we call the function `mv_qp`. (There is also a function named `minvar` in the `NMOF` package, which computes the minimum-variance portfolio.)

```
> mv_qp <- function(var, wmin, wmax) {
    na <- dim(var)[1L]
    A <- rbind(1, -diag(na), diag(na))
    bvec <- rbind(1,
                  array(-wmax, dim = c(na, 1L)),
                  array( wmin, dim = c(na, 1L)))
```

[mv-qp]

<sup>14</sup>You may also notice that we have chosen a fairly long historical window of about 10 years. We have about 50 assets, and so we end up with two and a-half times as many observations to compute a variance–covariance matrix. This choice is purely for expositional purposes here; practically, we might as well have chosen a shorter window. The trade-off is between bias and variance: if we use a shorter window, we will pick up more recent variation more quickly, but at the price of less-stable estimates. Choosing a longer window will make estimates more stable, but not necessarily more accurate: they may be more biased. Specifically for the dataset here, Fama and French (1997) report that factor loadings are varying strongly over time. A longer time horizon will produce more stable numbers, but these numbers are not necessarily better forecasts of future variance.

```
quadprog::solve.QP(
  Dmat = 2*var,
  dvec = rep(0, na),
  Amat = t(A),
  bvec = bvec,
  meq = 1L)$solution
}
```

The `mv_qp` function is not connected to `btest`; we may conveniently use it for other cases. And we may test it. For instance, let us compute the long-only MV portfolio for the first 5 years (approx. business 1250 days) of the data.

```
[qp-test] > library("quadprog")
> R <- returns(head(P, 1250), period = "month")
> var <- cov(R)
> w <- mv_qp(var, wmin = 0, wmax = 0.2)
```

As a plausibility check, we look at the 10 sectors with the lowest vol: they make up more than half the portfolio.

```
> df <- data.frame(vol = apply(R, 2, sd),
  weight = round(100*w, 2))
> sum(df[order(df$vol)[1:10], ])
```

```
[1] 55.6
```

## Computing the minimum-variance backtest

We run the backtest. Results are shown in Fig. 26. We rebalance at the end of each quarter. To compute the covariance, we use R's `cov` function. In an actual application, such a choice should be a point of criticism. There is ample empirical evidence that the forecasts of variances may be improved; for instance, by way of shrinkage methods. Thus, more interesting would be to test alternative functions, such as those in packages `robustbase` (Maechler, Rousseeuw, Croux, Todorov, Ruckstuhl, Salibian-Barrera, Verbeke, Koller, Conceicao, and Anna di Palma, 2018), `RiskPortfolios` (Ardia, Boudt, and Gagnon-Fleury, 2017) or `BurStFin` (Burns, 2014). But as we said at the start of the section, the main purpose will be to compare an exact method with a heuristic, for given inputs.

```
[bt-mv] > bt.mv <- btest(prices = list(coredata(P)),
  signal = signal_mv,
  do.signal = "lastofquarter",
```

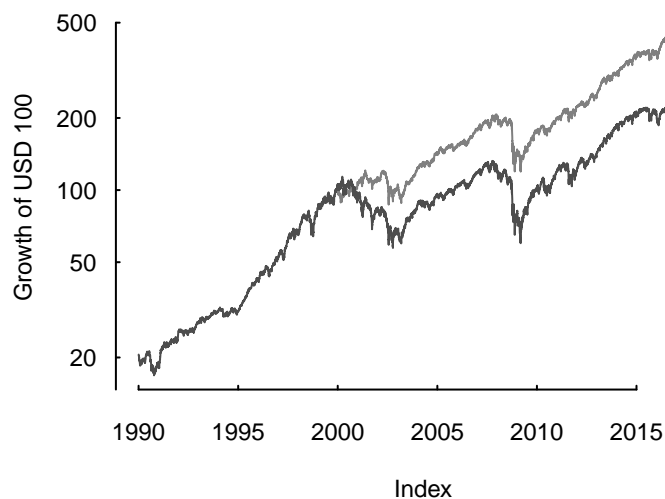


Figure 26: Performance of the long-only minimum-variance portfolio (gray) vs market (black). On Nov 19, 1999, both series have a value of 100 (the MV time series starts later because of its burn-in of 10 years).

```
convert.weights = TRUE,
initial.cash = 100,
b = 2500,
cov_fun = cov,
mv_fun = mv_qp,
wmax = 0.2,
wmin = 0.00,
n = 20,
timestamp = timestamp,
instrument = instrument)
```

Summary statistics for market and `bt.mv` summary may be computed by coercing both to `NAVseries` and calling `summary`. The summary statistics show that MV had a higher return (which is already visible from the figure) together with a lower volatility and a lower drawdown. This result is in line with other papers that analyzed low-risk portfolios in the US equity market, such as Chan, Karceski, and Lakonishok (1999) and Clarke, de Silva, and Thorley (2006).

```
> print(summary(window(as.NAVseries(market, title = "Market"),
                           start = as.Date("1999-12-31"))),
          sparkplot = FALSE, monthly.returns = FALSE)
```

[summaries]

```
-----
Market
31 Dec 1999 ==> 31 Jul 2018   (4675 data points, 0 NAs)
      515.676           1522.07
```

```
-----
High                1540.22  (25 Jul 2018)
Low                 278.65  (09 Oct 2002)
-----
```

```
Return (%)          6.0  (annualised)
-----
```

```
Max. drawdown (%)   54.7
_ peak              643.82  (09 Oct 2007)
_ trough            291.74  (09 Mar 2009)
_ recovery                      (13 Mar 2012)
_ underwater now (%)  1.2
-----
```

```
Volatility (%)      14.9  (annualised)
_ upside            10.8
_ downside          10.4
-----
```

```
> print(summary(window(as.NAVseries(bt.mv, title = "Minimum
  Variance"),
                      start = as.Date("1999-12-31"))),
  sparkplot = FALSE, monthly.returns = FALSE)
```

```
-----
Minimum Variance
31 Dec 1999 ==> 31 Jul 2018  (4675 data points, 0 NAs)
      100          486.064
-----
```

```
High                514.90  (26 Jan 2018)
Low                 86.95  (23 Jul 2002)
-----
```

```
Return (%)          8.9  (annualised)
-----
```

```
Max. drawdown (%)   42.3
_ peak              207.02  (10 Dec 2007)
_ trough            119.39  (09 Mar 2009)
_ recovery                      (04 Apr 2011)
_ underwater now (%)  5.6
-----
```

```
Volatility (%)      11.6  (annualised)
_ upside            9.0
_ downside          7.8
-----
```

## Sensitivity

As we pointed out, sensitivity analysis is one key job when doing backtests. Many parameters and settings could and should be checked for the described backtest. One such setting is the method of computing the variance–covariance matrix, and also the way in which the data were prepared. We passed a parameter `n`, so we may check what influence comes from choosing the holding period for returns. Important is of course the model and its restrictions: imposing minimum and maximum weights may have much influence on the results. Another parameter is the historical window, which we have chosen quite long. That brings the advantage that the computed variance–covariance matrix is fairly stable, at the price of the matrix not, or only slowly, picking up recent changes in the market.

Let us repeat the sensitivity check we did with the momentum strategy. We rebalance at randomly chosen intervals between one week (5 days) and about half a year (125 days).

```
> library("parallel")
> runs <- 100
> args_rnd_rebalance <- vector("list", length = runs)
> for (i in seq_len(runs)) {
  when <- cumsum(c(2501, sample(5:125, 5000, replace=TRUE)))
  when <- when[when <= length(timestamp)]

  args_rnd_rebalance[[i]] <-
    list(prices = list(coredata(P)),
         signal = signal_mv,
         do.signal = when,
         convert.weights = TRUE,
         initial.cash = 100,
         cov_fun = cov,
         mv_fun = mv_qp,
         wmax = 0.1,
         wmin = 0.00,
         n = 20,
         b = 2500,
         timestamp = timestamp,
         instrument = instrument)
}
> cl <- makePSOCKcluster(rep("localhost", 4))
> clusterEvalQ(cl, require("PMwR"))
```

[when-rebalance]

```
[[1]]
```

```
[1] TRUE
```

```
[[2]]  
[1] TRUE
```

```
[[3]]  
[1] TRUE
```

```
[[4]]  
[1] TRUE
```

```
> variations_rnd_rebalance <-  
  clusterApplyLB(cl, args_rnd_rebalance,  
                function(x) do.call(btest, x))  
> stopCluster(cl)  
> series_rnd_rebalance <- do.call(merge_series,  
                                variations_rnd_rebalance)
```

As Fig. 27 shows, there is quite some variation in the results, which becomes more salient when we compute actual numbers.

```
> cat("Annualized volatilities along price paths:\n")
```

```
Annualized volatilities along price paths:
```

```
> summary(16*apply(returns(series_rnd_rebalance), 2, sd))
```

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
0.147	0.148	0.149	0.149	0.149	0.151

```
> cat("Annualized returns along price paths:\n")
```

```
Annualized returns along price paths:
```

```
> summary(returns(series_rnd_rebalance, period = "ann"))
```

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
0.0852	0.0914	0.0959	0.0957	0.0995	0.1068

We see that realized volatilities vary in a range of half a percentage point, which does not seem too much in the equity world. But annual returns vary by more than two percentage points, and that is substantial. (And of course,

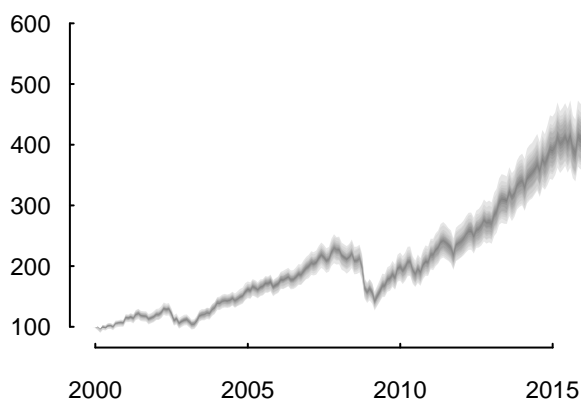


Figure 27: Performance of 100 walk-forwards with random rebalancing periods. The portfolios are computed with QP. All randomness in the results comes through differences in the setup; there is no numeric randomness.

we ran only 100 paths. With more paths, the range would almost certainly widen.) This variation, we should stress, appears because of different drift parameters. The chosen portfolios are similar: in Fig. 28, we plot the daily returns of the first four rebalancing variations. Their correlations are close to 1. That should not be surprising, as the actual positions are very similar, as Fig. 29 shows for a single asset.

Let us stress it: the variability in outcomes is entirely caused by variations in our settings; more specifically, the choice of when to rebalance. All computations are strictly deterministic; there is no chance involved.

## Using Local Search

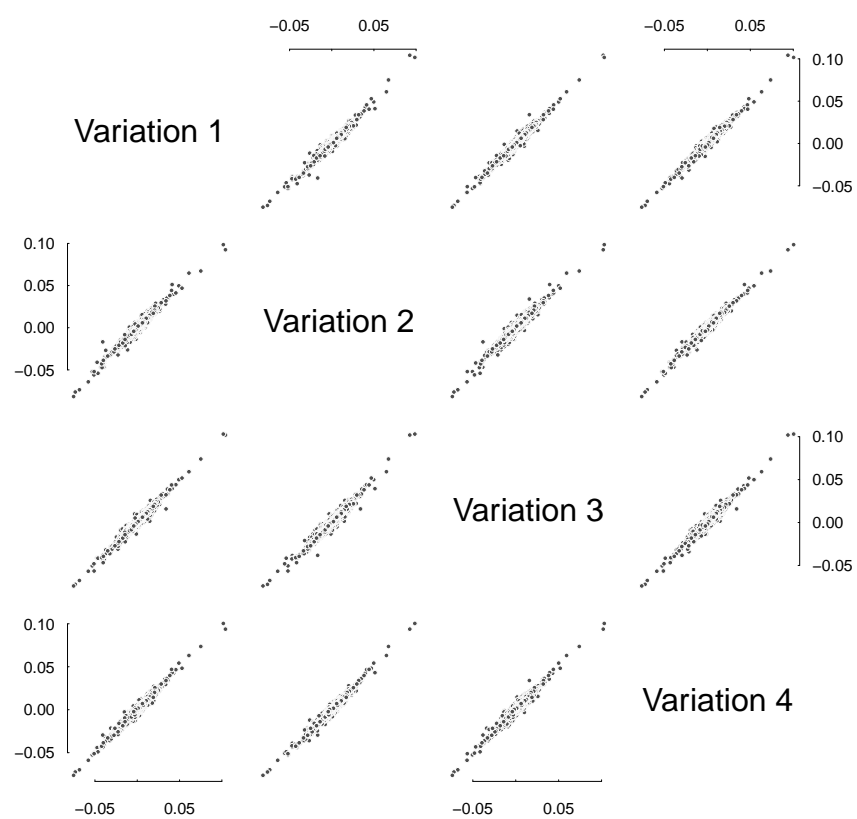
A second MV function, `mv_ls`, which uses Local Search.

```
> mv_ls <- function(var, wmin, wmax) {                                     [mv_ls]

  na <- dim(var)[1L]
  if (length(wmin) == 1L)
    wmin <- rep(wmin, na)
  if (length(wmax) == 1L)
    wmax <- rep(wmax, na)

  .neighbour <- function(w) {
    stepsize <- runif(1L, min = 0, max = 0.1)
    toSell <- which(w > wmin)
    toBuy <- which(w < wmax)
    i <- toSell[sample.int(length(toSell), size = 1L)]
  }
```

Figure 28: Correlation of daily returns when portfolios are at random timestamps.





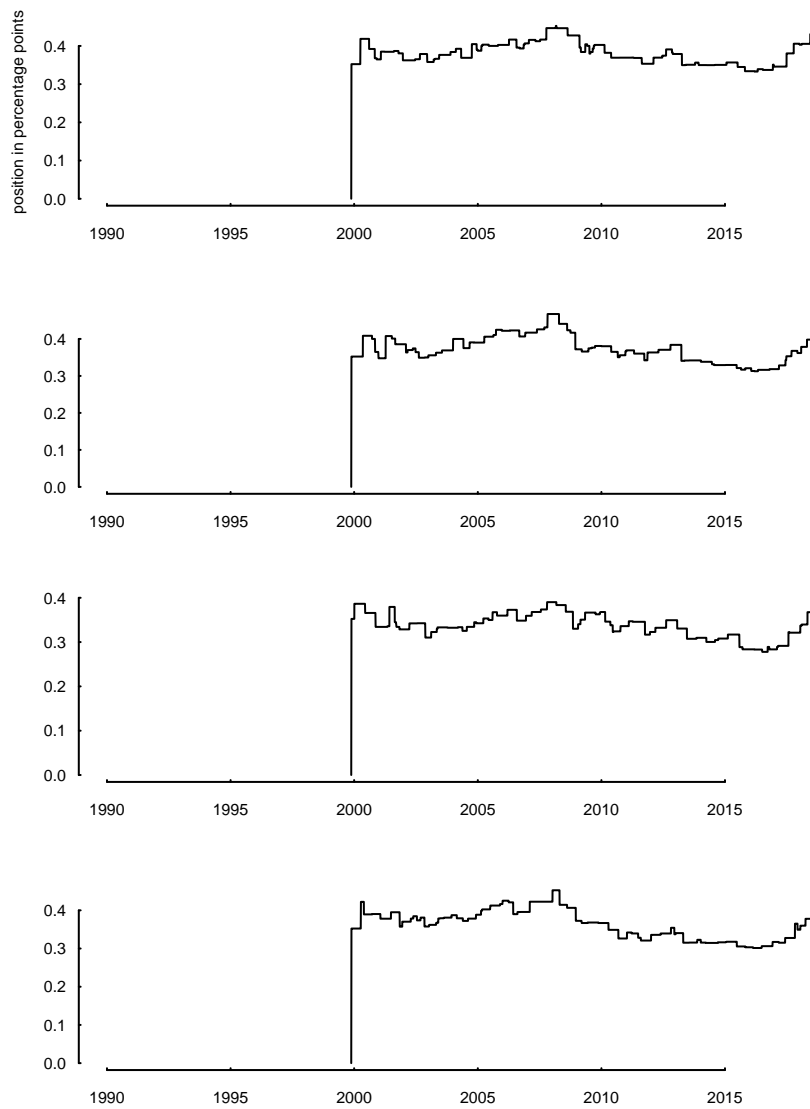


Figure 29: Positions in a single industry across four backtest variations. The backtests differ only in their rebalancing schedules; hence the positions are very similar.

```

        j <- toBuy[sample.int(length( toBuy), size = 1L)]
        stepsize <- runif(1) * stepsize
        stepsize <- min(w[i] - wmin[i], wmax[j] - w[j],
                        stepsize)
        w[i] <- w[i] - stepsize
        w[j] <- w[j] + stepsize
        w
    }

    .pvar <- function(w)
        w %*% var %*% w

    NMOF::LSopt(.pvar,
                list(neighbour = .neighbour,
                     x0 = rep(1/na, na),
                     nI = 10000,
                     printBar = FALSE,
                     printDetail = FALSE))$xbest
}

```

We compare the differences in weights, in basis points.

```

> w.qp <- mv_qp(var, 0.0, 0.2)
> w.ls <- mv_ls(var, 0.0, 0.2)
> summary(10000*(w.qp - w.ls))

```

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
-2.06	0.00	0.00	0.00	0.00	4.54

For the most part, they are very small, with some outliers. (But keep in mind that we may invest up to 10% into a single sector, so 10 basis points is not much.) Let us see the impact of these deviations. We run 100 backtests with `mv_ls`.

```

[variations-ls] > variations_ls <-
                btest(prices = list(coredata(P)),
                      signal = signal_mv,
                      do.signal = "lastofquarter",
                      convert.weights = TRUE,
                      initial.cash = 100,
                      b = 2500,
                      cov_fun = cov,
                      mv_fun = mv_ls,
                      wmax = 0.2,

```

```
wmin = 0.00,
n = 20,
timestamp = timestamp,
instrument = instrument,
replications = 100,
variations.settings =
  list(method = "parallel",
        cores = 10))
```

The results are shown in Fig. 30. There is variation in the paths, but it is so tiny that it is barely visible in the figure.

```
> series_variations_ls <- do.call(merge_series, variations_ls)
> cat("Annualized volatilities along price paths:\n")
```

```
Annualized volatilities along price paths:
```

```
> summary(16*apply(returns(series_variations_ls), 2, sd))
```

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
0.146	0.146	0.146	0.146	0.146	0.146

```
> cat("Annualized returns along price paths:\n")
```

```
Annualized returns along price paths:
```

```
> summary(returns(series_variations_ls, period = "ann"))
```

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
0.0881	0.0881	0.0882	0.0882	0.0882	0.0883

## 5 Prices in **btest**

The 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 a list of four matrices with two columns each:

```
1 open      high      low      close
```

The line graph illustrates the annual number of U.S. patents granted from 2000 to 2016. The vertical axis (y-axis) measures the number of patents, ranging from 100 to 500 in increments of 100. The horizontal axis (x-axis) shows the years from 2000 to 2016, with major ticks every five years. The data shows a steady increase from approximately 100 patents in 2000 to about 200 in 2008. A sharp decline occurs in 2009, where the number of patents drops to around 140. Following this dip, there is a rapid and consistent rise, reaching nearly 500 patents by 2016.

Year	Number of Patents (Approximate)
2000	100
2001	110
2002	120
2003	110
2004	130
2005	140
2006	150
2007	180
2008	200
2009	140
2010	180
2011	210
2012	240
2013	280
2014	320
2015	380
2016	480

2	+-+ -+	+ -+ -+	+ -+ -+	+ -+ -+
3				
4				
5				
6				
7				
8	++-+ -+	+ ++-+ -+	+ -+ -+	+ -+ -+

## A Notes on zoo

```
[zoo] > mom.latest <- mom(tail(coredata(P), 250), 10)
```

```
> mom(tail(P, 250), 10)
```

```
[1] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[26] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

The matching of timestamps happened when we used `/`. If you look into `Ops.zoo`, you would see this:

```
> merge(e1, e2, all = FALSE, retclass = NULL)
> NextMethod(.Generic)
```

In other words, before `zoo` does any of the elementary operations, it will merge the operands on the time-series, and drop those observations with NA values. In many circumstances this is extremely convenient behavior. Suppose for instance you wanted to compute the P/E ratio for a stock, and lack data for certain periods. But it may cause surprises at times. A simple example may make the behavior clearer.

```
> zoo(1, 1) + zoo(1, 5)
```

[zoo-merge]

```
Data:
numeric(0)
```

```
Index:
numeric(0)
```

```
> zoo(1, 1:2) + zoo(1, 2:3)
```

```
2
2
```

Since we are already at it: another gotcha when using `zoo` is integer timestamps.

```
> zoo(1, 5:10)
```

[zoo-integer]

```
 5  6  7  8  9 10
1  1  1  1  1  1
```

```
> zoo(1, 5:10)[1]
```

```
5
1
```

```
> zoo(1, 5:10)[5]
```

```
9  
1
```

```
> zoo(1, 5:10)[I(5)]
```

```
5  
1
```

## B Parallel Computations in R

### B.1 Distributed computing

In this section, we discuss some of the basics of parallel computations with package `parallel`. In fact, we will restrict the discussion to one subset of parallel computation, namely distributed computing, and we will stay in the context of backtesting.

Distributed computing is straightforward: split a large computation into smaller ones, and distribute these subcomputations to several workers; then, collect the results from the workers and combine them. Perhaps the simplest example is computing the sum of many numbers. Group the numbers into several subsets, give these subsets to workers that then compute the subset-sums; and, finally, add the subset-sums. The example already makes clear a trade-off: we save time because several workers do their job in parallel; but we lose time when we distribute the tasks, and collect and combine the results. The effort required for such “administrative” operations is called overhead. For computations that take very little time on a modern computer (such as, incidentally, computing a sum of numbers), distributing does not help because the overhead is too large. As a rule: the best way to find out whether a parallel computation actually saves time (and if so, how much) is to run experiments.

Distributed computing has several advantages: it is simple, and often scales well: if the overhead is small compared with the time the actual computation requires, the speedup is basically linear. Double the number of workers, and the computation time halves.

Cleve Moler called easily distributable computations “embarrassingly parallel”; such computations are everywhere in finance:

- Monte Carlo simulations;
- pricing portfolios in which the positions do not depend on one another;
- optimization with population-based methods;

- running restarts for optimization methods;
- general sensitivity analysis.

## B.2 Loops and **apply** functions

Let us define a simple function, `one`, which gives us what its name promises.

```
> one <- function(...)
  1
```

[one]

```
> one()
```

```
[1] 1
```

Suppose `one` actually did something useful, and you wanted to repeat this computation 1000 times. The reflex is to use a loop.

```
> runs <- 1000
> ones <- numeric(runs)
> for (i in seq_len(runs))
  ones[i] <- one()
```

[one-loop]

But a loop misguides us, in a way, since it implies an iterative computation: the first, the second, etc. But we care not about the order in which `one` is called.

In R, we may use `lapply` instead (or a higher level variant, such as `replicate`).

```
> ones <- lapply(seq_len(runs), one)
```

It is not merely a change in syntax: `lapply` does not make a promise about the order in which `one` is called.

Package `parallel` offers several parallel equivalents to `lapply`; one of them is called `parLapply`. Before we use the function, let us make `one` slower.

```
> one <- function(...) {
  Sys.sleep(1)  ## wait one second
  1
}
```

[one-wait]

Running `one` four times should now take just as many seconds (plus a little overhead).

```
> runs <- 4
> system.time(
```

[runs]

```
for (i in seq_len(runs))
  one())
```

```
user  system elapsed
0.004  0.000  4.009
```

The same is true with `lapply`.

```
> system.time(
  lapply(seq_len(runs), one))
```

```
user  system elapsed
0.00  0.00  4.01
```

But not when run in parallel, on four cores.

```
[one-parallel] > cl <- makeCluster(4) ## four cores
> system.time(parLapply(cl, seq_len(runs), one))
```

```
user  system elapsed
0.002  0.000  1.003
```

```
> system.time(clusterApply(cl, seq_len(runs), one))
```

```
user  system elapsed
0      0      1
```

```
> stopCluster(cl)
```

Note that when we call `makeCluster`, we assume that your machine has four cores. You may use function `detectCores` to find out about your machine. But be sure to read the functions help page and its caveats.

## B.3 Distributing data

Running a computation in parallel may always be split into three parts: distribute data and code to the nodes; have the nodes run the computations; and finally collect the results. R will help us with the second and the third task; in fact, it will do the whole job for us. That means we are left with the task of organizing the computation and distributing it.

Let us create a new function.

```
[sum-xy] > sum_xy <- function(x, y)
```



```
x + y
```

Suppose we want to evaluate the function for different values of `x` and `y`. These different values are collected in a data frame `df`.

```
> df <- expand.grid(x = 1:2, y = 5:6)
> df
```

```
  x y
1 1 5
2 2 5
3 1 6
4 2 6
```

In a serial computation, i.e. with a loop, we would run through the rows of `df` and call `sum_xy` for each row.

In R we may “package” the arguments of a function into a list, and then call the function with this single list as an argument. (In case you do not realize it: this is a very powerful feature.)

```
> args <- list(x = 1, y = 5)
> do.call("sum_xy", args)
```

[args-do-call]

```
[1] 6
```

The elements in `args` will be treated and matched as in a standard function call. So for instance `args <- list(1, x = 2)` means that `y` gets the value of 1.

A simple strategy is the one we described before: we pack every row of `df` into a list.

```
> data <- vector("list", length = nrow(df))
> for (i in seq_len(nrow(df)))
  data[[i]] <- list(x = df$x[i], y = df$y[i])
```

[pack-df]

<sup>15</sup> It is easy to memorize this pattern of setting up the data, because it resembles calling the function of interest in a loop, only instead of calling the function, we call `list`.

We could now call `lapply`.

```
> lapply(data, function(z) do.call(sum_xy, z))
```

[eval-lapply]

```
[[1]]
```

---

<sup>15</sup>It would have sufficed to write `data[[i]] <- c(df[i, ])` in the loop. Calling `c` has the (documented) side effect of dropping all attributes, including `class`.

```

[[1]]
[1] 6

[[2]]
[1] 7

[[3]]
[1] 7

[[4]]
[1] 8

```

We are as well ready for the parallel version of `lapply`. Or almost, at least: we have prepared the data, but the nodes are “clean”; they cannot know what `sum_xy` is. So we tell them, by exporting the function `sum_xy` from the master (i.e. the session that starts the other processes) to the nodes.

```

[export-fun] > cl <- makeCluster(4)
> clusterExport(cl, "sum_xy")
> parLapply(cl, data, function(x) do.call(sum_xy, x))

```

```

[[1]]
[1] 6

[[2]]
[1] 7

[[3]]
[1] 7

[[4]]
[1] 8

```

```

> clusterApply(cl, data, function(x) do.call(sum_xy, x))

```

```

[[1]]
[1] 6

[[2]]
[1] 7

[[3]]
[1] 7

```

```
[[4]]  
[1] 8
```

```
> stopCluster(cl)
```

Alternatively, we could have sent the code of `sum_xy` as an expression and evaluate it on each node.

```
> cl <- makeCluster(4)  
> ignore <- clusterEvalQ(cl,  
                           sum_xy <- function(x, y)  
                             x + y)  
  
> parLapply(cl, data, function(x) do.call(sum_xy, x))
```

[eval-fun]

```
[[1]]  
[1] 6  
  
[[2]]  
[1] 7  
  
[[3]]  
[1] 7  
  
[[4]]  
[1] 8
```

```
> stopCluster(cl)
```

## B.4 Distributing data, continued

In the example, all arguments to the computation were variable, i.e. they changed in every call. But suppose that `y` is fixed. In a backtest, think of price data, which may stay the same over different tests. Instead of moving such fixed data around every time, we might as well export it from the master to the nodes.

```
> y.value <- 100  
> x.values <- as.list(1:4)  
  
> cl <- makeCluster(4)  
> clusterExport(cl, "y.value")
```

[export-y]

```

> ignore <- clusterEvalQ(cl,
                           sum_xy <- function(x, y = y.value)
                               x + y)

> parLapply(cl, x.values, function(x) sum_xy(x, y.value))

```

```

[[1]]
[1] 101

```

```

[[2]]
[1] 102

```

```

[[3]]
[1] 103

```

```

[[4]]
[1] 104

```

```

> stopCluster(cl)

```

Let us create two tiny case studies for distributing a backtest. In one, the price data stays unchanged, but we wish to test different parameters. In the other, we wish to run the same strategy on two different data sets.

[backtests-parallel1]

```

> ## set up data, functions
> prices <- 101:110
> signal <- function(threshold) {
  if (Close() > threshold)
    1
  else
    0
}
> threshold.values <- as.list(102:105)

> ## create cluster and distribute data, functions
> cl <- makeCluster(4)
> clusterExport(cl,
                c("signal", "prices"))

> ignore <- clusterEvalQ (cl,
                          library("PMWR"))

```

```

> ## run btest
> parLapply(cl, threshold.values,
            function(x)
              btest(prices = prices,
                    signal = signal,
                    threshold = x))

```

```

[[1]]
initial wealth 0  =>  final wealth  6

[[2]]
initial wealth 0  =>  final wealth  5

[[3]]
initial wealth 0  =>  final wealth  4

[[4]]
initial wealth 0  =>  final wealth  3

```

```

> stopCluster(cl)

```

The second example.

```

> ## set up data, functions
> prices <- list(prices1 = 101:110,
                 prices2 = 201:210)
> signal <- function() {
  if (Close() > 105)
    1
  else
    0
}

> cl <- makeCluster(4) ## create cluster

> clusterExport(cl,      ## distribute data, functions
               c("signal"))
> ignore <- clusterEvalQ (cl,
                          library("PMwR"))

```

[backtests-  
parallel2]

```
> parLapply(cl, prices, ## run btest
           function(x)
             btest(prices = x,
                   signal = signal))
```

```
$prices1
initial wealth 0 => final wealth 3

$prices2
initial wealth 0 => final wealth 8
```

```
> stopCluster(cl)
```

As the examples have shown, there is typically more than one way to do it. In general, in particular for larger studies, it pays off to take some time to structure the computations and results, and to experiment with different setups.

One useful idea is for instance to store different specifications of backtests in files. If these are R code files, we may then use the `parallel` functions directly with `source`. Suppose for each strategy you have an R file in a directory `Backtesting`; then you could evaluate each file with a code snippet as this one.

```
[eval-files] > files <- dir("~/Backtesting",
                    pattern = "^.*\\.R",
                    full.names = TRUE)
> cl <- makeCluster(4)
> clusterApplyLB(cl = cl, files, source)
> stopCluster(cl)
```

## B.5 Other functions in package `parallel`

For an overview of the functionality of the `parallel` package, see the vignette that comes with the package. Since the package is recommended, it is usually installed by default. You can open the vignette directly from within R.

```
> vignette("parallel", package = "parallel")
```

## B.6 Parallel computations in the `NMOF` package

Several functions in the `NMOF` package have support for distributed computations built in. As of package version 1.5-0, these are `GAopt`, `gridSearch`,

bracketing and restartOpt.

## References

- David Ardia, Kris Boudt, and Jean-Philippe Gagnon-Fleury. RiskPortfolios: Computation of risk-based portfolios in R. *Journal of Open Source Software*, 10(2), 2017. doi: 10.21105/joss.00171.
- Hendrik Bessembinder. Do stocks outperform treasury bills? forthcoming.
- Marshall E. Blume and Robert F. Stambaugh. Biases in computed returns: An application to the size effect. 12(3), 1983.
- Patrick Burns. *BurStFin: Burns Statistics Financial*, 2014. URL <https://CRAN.R-project.org/package=BurStFin>. R package version 1.02.
- Louis K. C. Chan, Jason Karceski, and Josef Lakonishok. On portfolio optimization: Forecasting covariances and choosing the risk model. *Review of Financial Studies*, 12(5):937–974, 1999.
- Roger Clarke, Harindra de Silva, and Steven Thorley. Minimum-variance portfolios in the U.S. equity market. *Journal of Portfolio Management*, 33(1):10–24, 2006.
- Gilles Daniel, Didier Sornette, and Peter Wöhrmann. Look-ahead benchmark bias in portfolio performance evaluation. *Journal of Portfolio Management*, 36(1):121–130, 2009. URL <http://dx.doi.org/10.5167/uzh-6678>.
- Eugene F. Fama and Kenneth R. French. Industry costs of equity. *Journal of Financial Economics*, 43(2):153–193, 1997.
- Manfred Gilli and Enrico Schumann. Risk-reward ratio optimisation (revisited). <https://ssrn.com/abstract=2975529>, 2017.
- Manfred Gilli, Dietmar Maringer, and Enrico Schumann. *Numerical Methods and Optimization in Finance*. Elsevier/Academic Press, 2 edition. URL <http://nmof.net>.
- Manfred Gilli, Dietmar Maringer, and Enrico Schumann. *Numerical Methods and Optimization in Finance*. Elsevier/Academic Press, 2011. URL <http://nmof.net>.
- Benjamin Graham and David Dodd. *Security Analysis*. 1 (reprint) edition, 1934.

- Campbell R. Harvey, Yan Liu, and Heqing Zhu. ... and the cross-section of expected returns. *Review of Financial Studies*, 29(1):5–68, 2016. doi: 10.1093/rfs/hhv059. URL <http://dx.doi.org/10.1093/rfs/hhv059>.
- Narasimhan Jegadeesh and Sheridan Titman. Returns to buying winners and selling losers: Implications for stock market efficiency. *Journal of Finance*, 48(1):65–91, 1993.
- Martin Maechler, Peter Rousseeuw, Christophe Croux, Valentin Todorov, Andreas Ruckstuhl, Matias Salibian-Barrera, Tobias Verbeke, Manuel Koller, Eduardo L. T. Conceicao, and Maria Anna di Palma. *robustbase: Basic Robust Statistics*, 2018. URL <http://robustbase.r-forge.r-project.org/>. R package version 0.93-1.
- Tobias J. Moskowitz and Mark Grinblatt. Do industries explain momentum? *Journal of Finance*, 54(4):1249–1290, 1999. URL <http://www.jstor.org/stable/798005>.
- R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2018. URL <http://www.R-project.org/>.
- Enrico Schumann. *Portfolio Management with R*. 2008–2018a. URL <http://enricoschumann.net/PMwR/>.
- Enrico Schumann. *PMwR: Portfolio Management with R (Package version 0.10-0)*. 2008–2018b. URL <http://enricoschumann.net/PMwR/>.
- Enrico Schumann. *Numerical Methods and Optimization in Finance (NMOF) – Manual (Package version 1.5-0)*. 2011–2018. URL <http://enricoschumann.net/NMOF.htm#NMOFmanual>.
- Enrico Schumann. Two pitfalls in comparing financial time-series. available from <http://enricoschumann.net>, 2013.
- Robert J. Shiller. Price–earnings ratios as forecasters of returns: The stock market outlook in 1996, 7 1996. URL <http://www.econ.yale.edu/~shiller/data/peratio.html>.
- Robert J. Shiller. *Irrational Exuberance*. Princeton University Press, 2000.
- Robert J. Shiller. *Irrational Exuberance*. Princeton University Press, 3 edition, 2015.
- Achim Zeileis and Gabor Grothendieck. zoo: S3 infrastructure for regular and irregular time series. *Journal of Statistical Software*, 14(6):1–27, 2005.



# Index

- btest (R function)
  - Tutorial, 21–35
- burn-in, 22
- BurStFin (R package), 66
- Campbell, John, 48
- CAPE ratio, 37
- cashflow (R function), 26
- detectCores (R function in package parallel), 80
- Distributed computing, 78
- eqscplot (R function in package MASS), 7
- fan\_plot (R function), 52
- fanplot, 50–52
- French (R function in package NMOF), 48
- French, Kenneth, 48
- gridSearch (R function in package NMOF), 9
- Irrational Exuberance (book), 36
- MA\_crossover\_optimized (R function), 8
- makeCluster (R function in package parallel), 80
- MASS (R package), 7
- merge\_series (R function), 40
- minvar (R function in package NMOF), 65
- mom (R function), 56
- mv\_ls (R function), 71
- mv\_qp (R function), 65
- NMOF (R package)
  - parallel computing, 86
- one (R function), 79
- parallel (R package), 78
- parLapply (R function in package parallel), 79
- PMwR (R package), 21, 37
- quadprog (R package), 65
- R packages
  - BurStFin, 66
  - MASS, 7
  - PMwR, 21, 37
  - RiskPortfolios, 66
  - parallel, 78
  - quadprog, 65
  - robustbase, 66
  - zoo, 37
- rebalance
  - during backtest, 26
- replicate (R function), 79
- RiskPortfolios (R package), 66
- robustbase (R package), 66
- series\_ratio (R function), 41
- Shiller (R function in package NMOF), 36
- Shiller, Robert, 36
- source (R function), 86
- trade\_details (R function), 23
- Validation of backtest
  - single split, 15
  - walk-forward, 15
- Walk-forward, 15
- zoo (R package), 37