Portfolio Probe

Portfolio Probe User's Manual

Contents

1 Orientation **13**
- 1.1 Why Portfolio Probe? . . . 13
- 1.2 Overview of Functionality . . . 14
- 1.3 Necessary Tools . . . 14
- 1.4 Installing the Software . . . 15
- 1.5 Loading the Software . . . 15
- 1.6 Road Map . . . 15
- 1.7 Typography Conventions . . . 18

2 Generating Random Portfolios **19**
- 2.1 The Command . . . 19
- 2.2 Working with Random Portfolios . . . 21
  - Valuation . . . 21
  - Small Selections . . . 21
  - Evaluating Portfolios . . . 22
  - Summary . . . 23
- 2.3 Exporting Random Portfolios . . . 23
  - Writing monetary value . . . 24
- 2.4 Create a Matrix of Positions or Values . . . 24
- 2.5 Combining Random Portfolio Objects . . . 24
- 2.6 Unsatisfiable and Difficult Constraints . . . 25
- 2.7 Adding a Utility Constraint . . . 26
- 2.8 Going Farther . . . 26

3 Constraints **27**
- 3.1 Summary of All Constraints . . . 27
  - Round Lots . . . 27
- 3.2 Monetary Value of the Portfolio . . . 27
  - Long-only Portfolios . . . 29
  - Long-short Portfolios . . . 29
- 3.3 Limits on Assets . . . 31
  - max.weight . . . 31
  - universe.trade . . . 32
  - lower.trade and upper.trade . . . 32
  - risk.fraction . . . 33
- 3.4 Number of Assets . . . 36
  - Number of Assets to Trade . . . 36
  - Number of Assets in the Portfolio . . . 36

3.5 Threshold Constraints 37
Trade Thresholds 37
Portfolio Thresholds 37
Summary of Threshold Inputs 38
3.6 Forced Trades 38
3.7 Positions 38
Portfolio constraints 39
Trade constraints 40
Forced constraints 40
Universe constraints 41
Threshold constraints 41
Tolerance 41
Summary of positions inputs 42
3.8 Linear Constraints 42
Building Constraints 42
Bounds and lin.style 43
Linear Constraints on Variance Partitions 46
Numerical Constraints: Risk Factors 46
Numerical Constraints: Market Capitalization 47
Mixing Numerical and Categorical Constraints 48
Portfolio Constraints versus Trade Constraints 48
Net Constraints versus Gross Constraints 49
Long-side Constraints and Short-side Constraints 49
Looking at the Effect of the Constraints 50
Evaluating Un-imposed Constraints 51
Inspecting Linear Constraints 51
3.9 Count Constraints 52
3.10 Alpha (Expected Return) Constraints 55
3.11 Variance Constraints 55
3.12 Tracking Error (Benchmark) Constraints 56
Single Upper Bound 56
Scaling 56
Lower and Upper Bounds 56
Multiple Benchmarks 57
Advanced Use 57
3.13 Distance 57
Alternative prices 58
Multiple distances 58
3.14 Sums of Largest Weights 59
3.15 Cost Constraints 60
3.16 Number of Positions to Close 60
3.17 Quadratic Constraints 61
Add Constraints to the Variance 61
Impose Constraint Bounds 62
Dummy Run 62
Check for Benchmark 62
Constraints out of Utility 63
Actual Computation 63
3.18 Constraint Penalties and Soft Constraints 64

4 Valuation of Portfolios 65
4.1 Single Portfolio 65
Summary Statistics 66
Weights 66
Collapsing Values 67
4.2 Random Portfolios 67
4.3 Compute Returns 68
4.4 Collapse into Categories 68
4.5 Summary 69

5 Optimizing Long-Only Portfolios 71
5.1 Required Inputs 71
Monetary Value 71
Utility 71
5.2 Examples for Passive Portfolios 72
Minimize the Variance of the Portfolio 72
Minimize Tracking Error 72
5.3 Examples for Active Portfolios 74
Maximize the Information Ratio 74
The Information Ratio with a Tracking Error Constraint 74
Maximize Benchmark-relative Information Ratio 76
Mean-Variance Optimization 76
Mean-Volatility Optimization 76
Buy-Hold-Sell List 77
5.4 Utility-free Optimization 77
5.5 Managing Cash Flow 78
Injecting Money into a Portfolio 78
Extracting Money out of a Portfolio 79
5.6 Asset Allocation 79
5.7 Going Farther 80

6 Optimizing Long-Short Portfolios 81
6.1 Required Inputs 81
Monetary Value 81
Utility 81
6.2 Examples 82
Maximize the Information Ratio 82
Maximize Return with a Bound on the Variance 83
Minimize Variance Given a Long List and a Short List 83
Mean-Variance Optimization 84
6.3 Managing Cash Flow 85
Injecting Money into a Portfolio 85
Extracting Money out of a Portfolio 85
6.4 Money Constraints 86
6.5 Real-Time Monitoring 86
6.6 Going Farther 88

7 General Use **89**
7.1 Setting Up Data . . . 89
Prices and Other Imports . . . 89
Variance Matrix . . . 90
Adding a Benchmark to the Variance . . . 92
7.2 The Random Generation or Optimization . . . 92
7.3 Post-Optimization . . . 93
Explore the Trade . . . 93
Export the Trade . . . 94
7.4 Going Farther . . . 94

8 Trading Costs **95**
8.1 Background . . . 95
8.2 Specifying Costs . . . 96
Linear Costs . . . 96
Nonlinear Costs . . . 97
8.3 Power Laws . . . 98
8.4 On Scaling Costs Relative to Utility . . . 98
8.5 Costs Due to Taxes . . . 99
8.6 Going Farther . . . 100

9 Practicalities and Troubleshooting **101**
9.1 Easy Ways to Be Wrong . . . 101
Data Mangling . . . 101
Input Mangling . . . 103
9.2 Suppressing Warning Messages . . . 104
9.3 Cheatsheets . . . 107
Implied Ranges . . . 107
Threshold Inputs . . . 107
Positions Inputs . . . 108
9.4 Troubleshooting . . . 109
Utility Problems . . . 109
Portfolio Problems . . . 109
9.5 S Language Problems and Solutions . . . 110
Creating Matrices . . . 110
Debugging . . . 110

10 Special Instructions **113**
10.1 Special Instruction 1: Long-only when shorts exist . . . 113
10.2 Special Instruction 2: Benchmark in long-short optimization . . . 113

11 Adjusting Optimization Speed and Quality **115**
11.1 Staying at a Given Solution . . . 115
11.2 Reducing Time Use . . . 116
11.3 The Optimization Process . . . 117
11.4 Improving Quality . . . 118
11.5 Testing Optimization Quality . . . 118

12 Utility **121**
12.1 Maximum Information Ratio . . . 121
Example . . . 122
12.2 Mean-Variance Utility . . . 123
12.3 Mean-Volatility Utility . . . 123
12.4 Minimum Variance . . . 123
12.5 Maximum Expected Return . . . 124
12.6 Minimum Distance . . . 124
12.7 Going Farther . . . 124

13 Advanced Features **125**
13.1 Multiplicity . . . 125
13.2 Alpha and Variance Tables . . . 126
13.3 Variance Constraints . . . 127
13.4 Expected Return Constraints . . . 129
13.5 Multiple Utilities . . . 130
13.6 Utility Tables . . . 131
13.7 Multiplicity Examples . . . 132
Dual Benchmarks . . . 132
Benchmark-relative Utility and Absolute Variance Constraint . . 134
Rival Variance Forecasts . . . 135
Multiple Time Periods . . . 136
Credit Risk . . . 136
Multiple Scenarios . . . 138
13.8 Compact Variance Objects . . . 142
The Variance List . . . 143

14 Dregs **145**
14.1 Portfolio Probe Constituents . . . 145
14.2 The Objectives . . . 145
14.3 Writing C or C++ Code . . . 145
14.4 Bug Reporting . . . 146

List of Tables

3.1 Summary of constraints. 28
3.2 Choices for the `rf.style` argument. 35
3.3 Choices for the `lin.style` argument. 44

4.1 Summary of possibilities with `valuation`. 70
4.2 Explanation of the "results" column in Table 4.1. 70

9.1 Implied ranges of arguments. 108
9.2 The meaning of `threshold` inputs. 108
9.3 The column order for the `positions` argument. 108

13.1 Arguments for multiple variances and expected returns. 126
13.2 Utilities codes for the utility table. 132

List of Figures

1.1 Some suggested routes through the document. 16
1.2 Possible route for utility-free optimization. 17

6.1 Constraints on gross, net, long and short values. 87

Chapter 1

Orientation

This chapter has diverse aims:

- It suggests some reasons to choose Portfolio Probe.
- It provides a brief overview of Portfolio Probe functionality.
- It explains what software you need in order to run Portfolio Probe.
- It suggests a route through the rest of this document, given a task and a state of mind.
- It presents the typographic conventions of the document.

1.1 Why Portfolio Probe?

The ability to generate random portfolios efficiently is the biggest benefit of Portfolio Probe. Random portfolios are very powerful, and should be in your toolbox if you deal with portfolios.

There are some reasons you might choose Portfolio Probe over other software for optimization. These include:

- inputs and outputs of actual positions, not weights
- designed for long-short portfolios as well as long-only
- maximize information ratio directly
- can include riskless cash as an asset
- a wide range of utilities, including minimizing the distance to an ideal target portfolio
- integer constraints such as the number of names to trade, the number of names to hold, round lotting
- constraints on the risk contributed by each asset
- many other constraints

- flexible forms for transaction costs

Portfolio Probe is implemented in the S language, an environment specially designed for the type of operations that are needed when analyzing or optimizing portfolios.

1.2 Overview of Functionality

The two primary aims of Portfolio Probe are:

- To generate random portfolios (the `random.portfolio` function).
- To optimize a portfolio (the `trade.optimizer` function).

The two functions have the same inputs (except for saying how many random portfolios you would like, and whether you want random portfolios or random trades).

All of the rest of Portfolio Probe is support for these two tasks. The other major function is `valuation` that provides portfolio valuations or returns.

1.3 Necessary Tools

You need to choose a language in which to run Portfolio Probe. It can be one of three:

- R, which can be downloaded for free via:

  http://www.r-project.org/

There are some commercial distributions of R as well.

- S-PLUS (now S+), sold by TIBCO:

  http://spotfire.tibco.com/
- C++. You can call Portfolio Probe functionality in a program that you write.

S-PLUS and R are versions of the S language, and Portfolio Probe has been written to work with either version of S. This document assumes you are using S (as opposed to using C code). Portfolio Probe uses only very general features of S so it should run the same in any version of R or S-PLUS.

When this document says "S", it means either R or S-PLUS—the term "S" should not be construed to mean only S-PLUS. Some of the examples explicitly assume R is being used—the same effect would be done slightly differently in S-PLUS.

Programming experience is not mandatory—whatever your objective, there is likely to be an example provided in this manual or on the website that is close to your case.

The present document assumes knowledge of S to the level of "Some Hints for the R Beginner"—a brief, structured introduction which can be found in the

Tutorials section of http://www.burns-stat.com/. Commands beyond that level are included and explained.

The best way of using C++ is probably via the wonderful `RInside` package. See the cookbook section on the Portfolio Probe website for more details.

There is the possibility of directly calling the C code in Portfolio Probe. This approach is not recommended—it requires considerable effort, and likely has little or no benefit.

While it is reasonably easy to start using Portfolio Probe, there is a lot of room to grow. Portfolio Probe's flexibility and the power of S can carry you a long way.

1.4 Installing the Software

If you are using R and your machine has access to the internet, then you can install Portfolio Probe with:

```
> install.packages("PortfolioProbe",
+    repos="http://www.portfolioprobe.com/R")
```

See the Frequently Asked Questions in the "User area" of www.portfolioprobe.com if a firewall blocks this command or if you are using S-PLUS.

1.5 Loading the Software

If Portfolio Probe was installed in the default place, then Portfolio Probe is loaded into an R session with:

```
> library(PortfolioProbe)
```

It is possible that the `lib.loc` argument to `library` may be required. In S-PLUS it will depend on the particular installation, but something similar is likely.

1.6 Road Map

There is one of two frames of mind that you are likely to have:

- Conceptual: primarily wanting to understand the task
- Operational: primarily wanting to do the task

With the choice of two tasks, that produces four possible routes through the document. Figure1.1 is a graphical view of the suggestions.

Note that not all chapters appear in the maps. A lot of the document can be reserved for reference as the need arises. In particular if you are only generating random portfolios, you can safely ignore several of the chapters.

If you are only interested in utility-free optimization, then Figure 1.2 shows a possible route through this document. This supposes that you will impose a turnover constraint rather than providing trading costs for the assets.

Figure 1.1: Some suggested routes through the document.

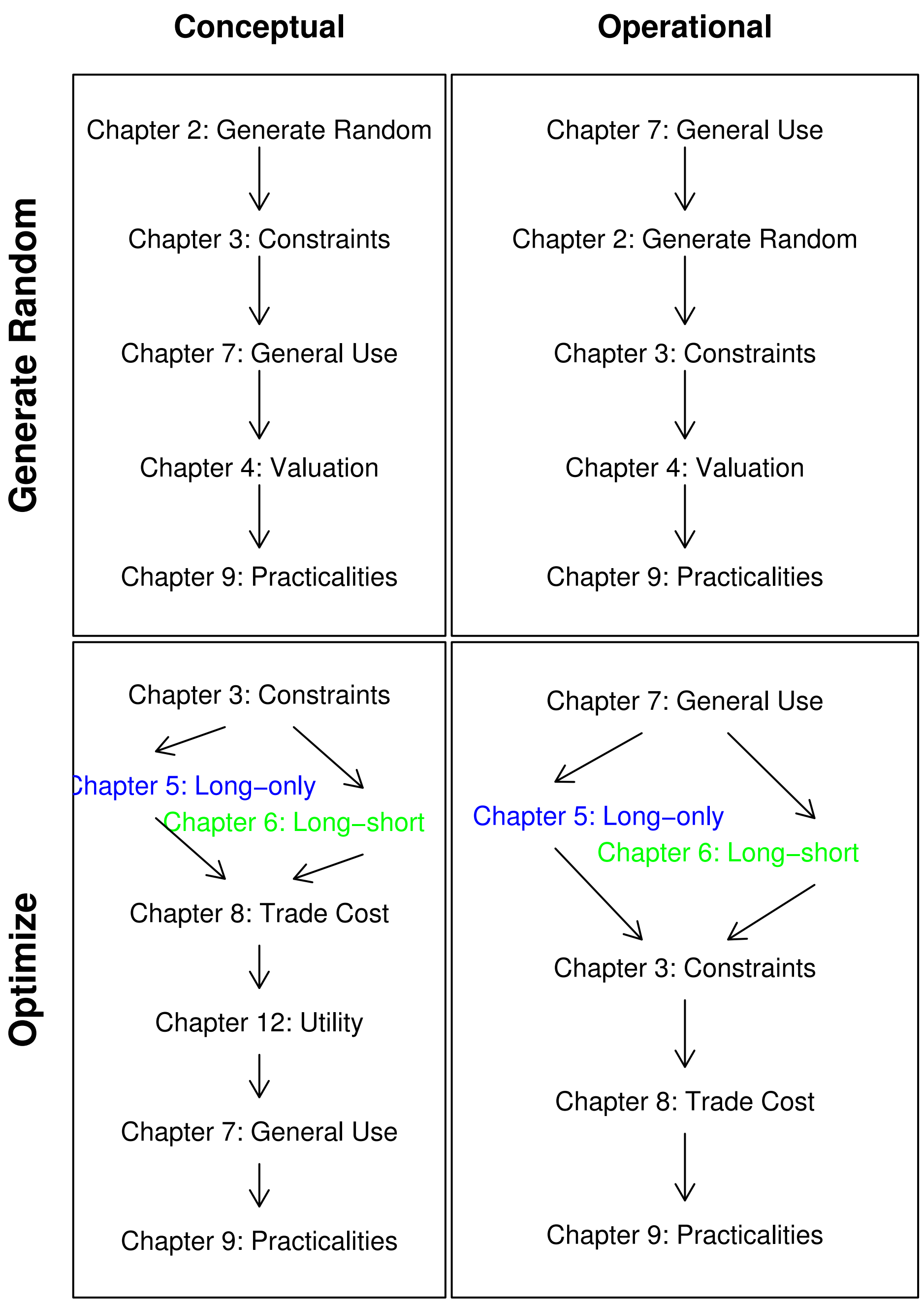
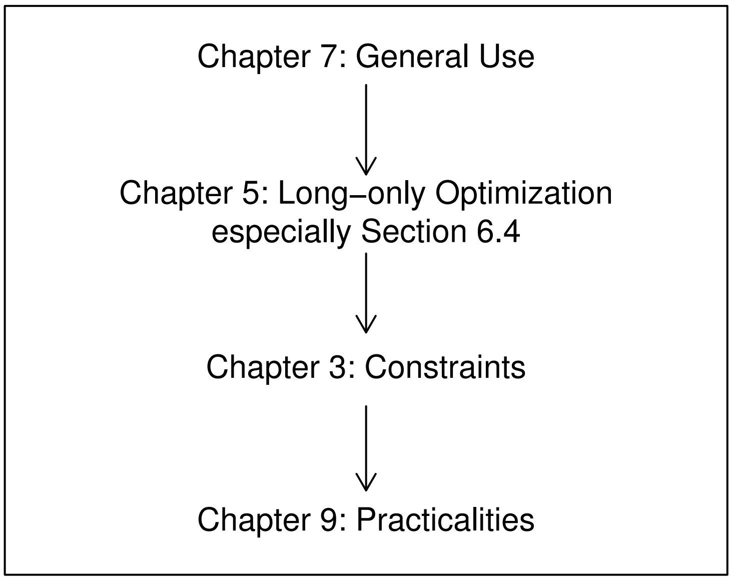


Figure 1.2: Possible route for utility-free optimization.



1.7 Typography Conventions

Computer commands or pieces of commands are written in `this font`. For example, a variance argument is written as `variance` whenever it is the argument itself that is referred to. Entire commands are written in the same font. Commands are introduced by "`> `" (which is the prompt in the S language) so that any output can be distinguished from the input. An example is:

```
> rep(0, 6)
[1] 0 0 0 0 0 0
```

The user types "rep(0, 6)" (followed by return), and the next line is the response from S.

Commands may span more than a single line—the second and subsequent lines of a command are introduced by "+ ". For example:

```
> op <- trade.optimizer(prices, varian, gross.value=1e6,
+    long.only=TRUE)
```

The second line of the command starts with `long.only` (the "+ " is not typed by the user, but rather by S). There is no output in this example.

S code note

The only catch with multi-line commands is that it needs to be clear to S that the command is incomplete. In this example the command needs a closing parenthesis.

Occasionally a fragment of code is written, in which case there are no introductory prompts.

In addition to S code notes, there are boxes which contain "cautions" and "notes".

Chapter 2

Generating Random Portfolios

The random.portfolio function generates a list of portfolios (or the trades for them) that satisfy constraints but pay no attention to utility.

2.1 The Command

To generate random portfolios you give random.portfolio the number of random portfolios that you want to generate, the basic information for the problem, and the constraints that you would like. There is also the out.trade argument which controls whether it is the random portfolio (the default) or the trade which is output.

At a minimum you need to specify the vector of prices and the amount of money in the portfolio. One possibility is:

```
> randport1 <- random.portfolio(prices=prices, long.only=TRUE,
+    gross.value=1e6)
```

Of course this is not very interesting. You are likely to want more than one random portfolio, and to have non-trivial constraints imposed.

S code note

The notation 1e6 means 1,000,000, that is, a one followed by six zeros.

The prices (always required) needs to be a vector of positive numbers that has names which identify the assets in the problem. Here is an example of the first few values of a suitable price vector:

```
> head(pricevec)
stockA stockB stockC stockD stockE stockF
 27.63  19.46  11.67   5.79   5.15  20.99
```

The assets named in `prices` define the universe of assets for the problem. In examples the `prices` argument is often given a vector called `prices`—in reality the name of the vector can be whatever you like.

The other two pieces of "basic information" are the variance matrix and the vector of expected returns—neither of these are required. You only need to give these if they are involved in a constraint.

To generate 100 random portfolios that have country and sector constraints, no more than a 4% tracking error and no more than 55 assets, the following command would do:

```
> randport2 <- random.portfolio(100, prices, varian,
+    long.only=TRUE, gross.value=1e6,
+    bench.constraint = c(spx=.04^2/252),
+    port.size=55, lin.constraints=cntrysect.constraint,
+    lin.bounds=cntrysect.bounds)
```

S code note

The first three arguments in the call that creates `randport2` do not need to have the name of the argument specified because they are all in the order of the arguments in the definition of the function. In contrast the call that creates `randport1` uses the argument name in all cases. If there is any doubt, then it is safest to give the argument by name.

The examples so far assume that there is no existing portfolio (or that it doesn't matter). The `existing` argument gives the current portfolio.

```
> randport3 <- random.portfolio(100, prices, varian,
+    long.only=TRUE, gross.value=1e6,
+    bench.constraint = c(spx=.04^2/252),
+    existing=current.portfolio,
+    port.size=55, lin.constraint=cntrysect.constraint,
+    lin.bounds=cntrysect.bounds)
```

Sometimes it is more convenient to have the trades rather than the portfolios. If you want the trades, just set the `out.trade` argument to `TRUE`:

```
> randtrade3 <- random.portfolio(100, prices, varian,
+    long.only=TRUE, gross.value=1e6,
+    bench.constraint = c(spx=.04^2/252),
+    existing=current.portfolio,
+    port.size=55, lin.constraint=cntrysect.constraint,
+    lin.bounds=cntrysect.bounds, out.trade=TRUE)
```

S code note

The full name of the `out.trade` argument must be given. This is unlike almost all other arguments where only enough of the first portion of the name needs to be given to make it unique among the arguments to the function. (For a full explanation of argument matching in S, see [Burns, 1998] page 19 or [Burns, 2011].)

If the `existing` argument is not given or is `NULL`, then it doesn't matter which value `out.trade` has—the output is the same in either case.

The result of a call to `random.portfolio` is a list where each component of the list is a portfolio (or trade). The object has a number of attributes including a `class` attribute (`"randportBurSt"`). Here is a small example:

```
> random.portfolio(2, priceten, gross.value=1e5,
+    long.only=TRUE, port.size=3, max.weight=.5)
[[1]]
stockA stockB stockJ
  1337   1481   6484
[[2]]
stockB stockF stockH
  1274   2382   3100
attr(,"call")
random.portfolio(number.rand = 2, prices = priceten,
    gross.value = 1e+05,long.only = TRUE, port.size = 3,
    max.weight = 0.5)
attr(,"timestamp")
[1] "Thu Mar 29 11:36:03 2012" "Thu Mar 29 11:36:03 2012"
attr(,"class")
[1] "randportBurSt"
seed attribute begins: 1 -1142929704 1716596987 -285978235
```

Each component of the list is a portfolio (or trade), which is a vector giving the number of asset units (shares, lots, contracts) for each asset that appears.

2.2 Working with Random Portfolios

Valuation

The most likely thing to do with random portfolios is to get their valuation or returns. This is discussed in Chapter 4.

Small Selections

You can use `head` to get the first few random portfolios, and `tail` to get the last few. These are generic functions in R. Their random portfolio methods return an object that retains the class and other attributes of the original object.

These functions can be useful to inspect the portfolios to see if they look reasonable without printing hundreds or thousands of portfolios to the screen. They can also be used to test commands, such as the example immediately below.

Evaluating Portfolios

The sister function to `random.portfolio` is `trade.optimizer`. It can be of interest to see some of the values that the optimizer would return for each of the random portfolios. The `randport.eval` function does that: for each of the random portfolios (or trades) in the object it finds what the optimizer says about it. You can select which components of the output of `trade.optimizer` to keep (using the `keep` argument). The result is a list as long as the random portfolio object and each component of that list is a list containing the kept components.

Here is a small example of keeping the portfolio variances:

```
> randport.eval(head(randport4, 3), keep='var.values')
[[1]]
[[1]]$var.values
      V0
382.3576
[[2]]
[[2]]$var.values
      V0
147.6476
[[3]]
[[3]]$var.values
      V0
134.6368
```

In this case where we are returning only one number per portfolio, it makes more sense to coerce this to a numeric vector:

```
> unlist(randport.eval(head(randport4, 3), keep='var.values'),
+    use.names=FALSE)
[1] 382.3576 147.6476 134.6368
```

Keep in mind that these values are *ex ante* predictions—they may or may not have much relation to realized variance.

note

In `randport.eval` the optimizer is called using the same names of objects as was used when the random portfolio object was originally created. Objects with these names must be visible at the time that `randport.eval` is used. If any of these objects has changed, then it is the current value rather than the original value that is used.

caution

Additional arguments or changes to arguments may be given to `randport.eval` so that what the optimizer does is not exactly what `random.portfolio` did. If you are making a change to an argument, then you need to use the exact same abbreviation (if any) as in the original call to `random.portfolio`.

There is a FUN argument to randport.eval that, if given, applies that function to each of the portfolio objects that are created. For example, we could do:

```
> randport.eval(head(randport4, 3), FUN=summary)
```

Or perhaps a more useful command along the same lines:

```
> do.call("rbind", randport.eval(randport4,
+    FUN=function(x) summary(x)$number.of.assets))
```

S code note

The command:

```
> do.call("rbind", some.list)
```

is equivalent to the command:

```
> rbind(some.list[[1]], some.list[[2]], ...,
+     some.list[[length(some.list)]])
```

Summary

The summary method for random portfolios shows how many assets are in the portfolios, and the number of times each asset appears in a portfolio:

```
> summary(randport5)
$port.size
  7   8   9  10
  1  32 165 802
$count.assets
stockC stockD stockA stockB stockE stockF stockI
  1000   1000    987    975    973    972    972
stockG stockH stockJ
   968    961    960
```

This shows us that out of the 1000 portfolios, 802 contained all 10 assets, 165 had 9 assets, 32 had 8 assets and 1 had 7 assets. We also see that stockC and stockD were both in all of the portfolios while stockJ was only in 960 of them.

2.3 Exporting Random Portfolios

The deport function will write files containing the result of random.portfolio. The simplest use is:

```
> deport(randport2)
[1] "randport2.csv"
```

This writes a comma-separated file where the columns each correspond to one of the assets that appear in the object and the rows correspond to the portfolios or trades. There are arguments that allow you to switch the meaning of rows and columns, and to give a universe of assets (which must include all of those appearing in the object). See the help file for details.

Writing monetary value

If you want the file to represent money rather than the number of asset units, you can use the `multiplier` argument:

```
> deport(randport2, multiplier=prices, file="randval1")
[1] "randval1.csv"
```

2.4 Create a Matrix of Positions or Values

At times it is useful to have a matrix where columns represent assets and rows represent random portfolios. There is a slightly devious way of getting this using `deport`:

```
> randmat <- read.table(deport(randPortObj,
+    filename="rpfile", sep=","), sep=",", header=TRUE)
> randmat[is.na(randmat)] <- 0
> head(randmat)
  stockA stockB stockC stockD stockE
1     54     43     54    115    267
2     33     49    128     37    277
3     24     37    128    259    121
4     45     77    127    134      0
5     38     25     86    166    291
6     54     54    128      0    187
```

The string used as `filename` is immaterial except that you don't want to overwrite a file that you already have.

S code note

The object above called `randmat` is actually a data frame, not a matrix. Sometimes it matters which you have, sometimes not. The functions `as.matrix` and `as.data.frame` can be used to switch between them.

2.5 Combining Random Portfolio Objects

You may want to combine some random portfolio objects. Suppose you have objects named `rp1`, `rp2` and `rp3` resulting from calls to `random.portfolio`. You would like these to be in one object as they all have the same constraints (or perhaps they have slightly different constraints but you want them all in the same analysis). The `c` function will put them all together:

```
> rp.all <- c(rp1, rp2, rp3)
```

But not all is well:

```
> deport(rp.all)
Error in deport(rp.all) : no applicable method for "deport"
> summary(rp.all)
      Length Class  Mode
 [1,] 45     -none- numeric
 [2,] 45     -none- numeric
 [3,] 45     -none- numeric
 [4,] 45     -none- numeric
 ...
```

Even though `rp.all` is basically correct, it doesn't have the class that the other objects have. Without the class, generic functions like `summary`, `deport` and `valuation` don't work as expected.

```
> class(rp.all) <- class(rp1)
> deport(rp.all)
[1] "rp.all.csv"
```

Once the class is put on the object, we can operate as usual.

Almost. If you want to use `randport.eval`, then you need the `call` attribute as well. In that case, you could give the big object all of the attributes of one of the original objects:

```
> attributes(rp.all) <- attributes(rp1)
```

2.6 Unsatisfiable and Difficult Constraints

Not all sets of constraints can be achieved. Obviously there are no portfolios that satisfy a variance that is smaller than zero (or even smaller than the minimum variance given the other constraints). If you set `random.portfolio` such a task, it is bound to fail.

There is a trade-off between returning quickly when asked the impossible and being successful when asked the merely difficult. There is, of course, a default value for this trade-off, but you can adjust it for specific circumstances. There are a number of control arguments that say how hard to work.

In terms of impossible constraints, the most important is `init.fail`. This says how many separate attempts to make before quitting when there have been no portfolios successfully found.

For each attempt there is a trio of arguments that control how hard to work within the attempt. `iterations.max` gives the maximum number of iterations before stopping. `fail.iter` is the maximum number of consecutive iterations allowed that fail to make progress. `miniter` gives the minimum number of iterations allowed even if `fail.iter` says to quit. (Of course if a portfolio is found that satisfies all the constraints, then the attempt is declared successful and stops no matter what the value of `miniter`.)

The remaining argument of this ilk is `gen.fail`. Let's start with the problem that this argument solves. Suppose you have set a difficult problem for `random.portfolio` and you want 1000 portfolios. Suppose further that the first attempt was successful (so clearly the problem is not impossible) but the next one million attempts fail. As far as you are concerned you are waiting forever. `gen.fail` times the number of portfolios requested is the maximum number of failed attempts allowed.

In our example you requested 1000 and the default value of `gen.fail` is 4, so it would stop after 4000 failures and return the one random portfolio it successfully found (and warn you that it didn't do so well with your request).

Note that it is seldom obvious whether a specific set of constraints is easy to satisfy, difficult to satisfy or impossible.

2.7 Adding a Utility Constraint

The `random.portfolio` function does not allow a constraint on the utility, but `random.portfolio.utility` does. If computation time is of concern, then it can be better to just use `random.portfolio` with constraints on the variance and expected returns. However, this may not be quite what you want.

There is not much difference between the functions in terms of how they are used. The key difference is the `objective.limit` argument. The objective is the negative of the utility. So if you want the utility to be at least 0.6, then you want the argument:

```
objective.limit = -0.6
```

The meaning of `gen.fail` is the same, but the other control arguments are those used with optimization.

There are two forms of failure:

- the objective does not achieve `objective.limit`
- the objective is okay, but there are other broken constraints

The `objfail.max` argument controls how many of the first type are allowed. If `objfail.max=1` (the default) and the first try does not achieve the objective limit, then an error is triggered.

The calls look like:

```
> rp1 <- random.portfolio(100, the.prices, ...)
> ru1 <- random.portfolio.utility(100, -0.6, the.prices, ...)
```

2.8 Going Farther

Tasks that you might want to undertake:

- Chapter 3 discusses how to specify the constraints.
- Chapter 4 shows how to get valuations and returns.
- To review common mistakes, see Section 9.1 on page 101.

Chapter 3

Constraints

This chapter covers the constraints that can be imposed for generating random portfolios and for optimization. `random.portfolio` and `trade.optimizer` use the exact same set of constraint arguments.

3.1 Summary of All Constraints

Table 3.1 lists the possible constraints along with the arguments used to achieve them. In addition to these explicit constraints, there is the implicit constraint of trading integer numbers of asset units.

Round Lots

Trading is only done in integer numbers of asset units except when there is an existing position that is non-integral. Thus if the prices are given for lots as opposed to shares, then round lotting is automatic.

3.2 Monetary Value of the Portfolio

This section discusses the arguments:

- `gross.value`
- `net.value`
- `long.value`
- `short.value`
- `turnover`
- `long.only`
- `allowance`

Table 3.1: Summary of constraints.

Arguments	**Constraint**	**Section**
`gross.value` `net.value` `long.value` `short.value` `allowance`	monetary value of the portfolio	3.2
`turnover`	turnover (buys plus sells)	3.2
`long.only`	no short values if `TRUE`	3.2
`max.weight`	maximum weight in the portfolio per asset	3.3
`universe.trade`	restrict assets to be traded	3.3
`lower.trade` `upper.trade`	lower and upper bounds on the the number of asset units to trade	3.3
`risk.fraction` `rf.style` `rf.loc`	fraction of variance attributed to assets, also asset correlation with the portfolio	3.3
`ntrade`	number of assets to trade	3.4
`port.size`	number of assets in the portfolio	3.4
`threshold`	threshold constraints on trade and portfolio	3.5
`forced.trade`	trades that must be done	3.6
`positions` `tol.positions`	otherwise available constraints expressed in monetary terms	3.7
`lin.constraints` `lin.bounds` `lin.trade` `lin.abs` `lin.style` `lin.direction` `lin.rfloc`	linear constraints on the portfolio and/or the trade using weights, values, variances, counts	3.8 3.9
`alpha.constraint`	bound on expected return of the portfolio	3.10
`var.constraint`	bound on variance of the portfolio	13.3
`bench.constraint`	bound on squared tracking error	3.12
`dist.center` `dist.style` `dist.bounds` `dist.trade` `dist.utility` `dist.coef`	distance from one or more portfolios	3.13
`sum.weight`	max (and min) of the sum of a specified number of the largest weights	3.14
`limit.cost`	allowable range of costs	3.15
`close.number`	number of positions to close	3.16

While there is no single monetary argument that needs to be given, it is mandatory that the monetary value of the portfolio be constrained somehow.

All of the monetary arguments are in the currency units that `prices` uses.

In all cases it is sufficient to only give `turnover`. The argument:

```
turnover = 12000
```

says that the buys plus sells of the trade can not exceed 12,000 currency units.

While this makes most sense when there is an existing portfolio, that is not necessary. The turnover can, of course, be constrained even when other monetary constraints are given.

The turnover can be expressed as an interval:

```
turnover = c(11000, 12000)
```

When only one number is given, the implied lower bound is zero.

How the other monetary arguments are used largely depends on whether or not the portfolio is long-only.

Long-only Portfolios

If you want long-only portfolios, then you need to set the `long.only` argument to `TRUE` (the default is `FALSE`).

You can state the amount of money in the resulting portfolio by giving the `gross.value` argument. Ultimately this needs to be a range of allowable values. You can give the range explicitly with a vector of two numbers:

```
gross.value = c(999900, 1e6)
```

Alternatively you can give a single number:

```
gross.value = 1e6
```

When a single number is given, this is taken to be the upper bound—the lower bound is computed via the `allowance` argument. The default allowance is 0.9999, that is, one basis point away. So (by default) the above two specifications of `gross.value` are equivalent.

In general there is no problem with a constraint this tight—the key thing is how wide the range is relative to the prices of the assets. There will be a warning if the interval is seen to be too narrow.

In the case of long-only portfolios, `net.value` and `long.value` are synonyms for `gross.value`, so you can give any one of these three.

Long-short Portfolios

The arguments `gross.value`, `net.value`, `long.value` and `short.value` control the value of the portfolio.

To be clear: The long value is the amount of money in positive positions. The short value is the amount of money in negative positions—this is meant to be a positive number, but the absolute value of negative numbers is taken for the short value. The gross value is the sum of the long and short values. The net value is the long value minus the short value.

There are two minimal sets of these arguments:

- `gross.value` and `net.value`
- `long.value` and `short.value`

Here are some examples:

```
gross.value = 1e6, net.value = c(200, 3000) # OK
long.value = 6e5, short.value = 5e5 # OK
gross.value = 1e6, net.value = c(0, 2e5), long.value=6e5 # OK
gross.value = 1e6, long.value = 6e5 # not OK, neither pair
```

These four arguments are logically each of length two—giving allowable ranges. If they only have length one, then the second value is computed by multiplying the given value by the `allowance` argument. The default value for `allowance` is 0.9999, that is, one basis point away—you may want to alter this depending on how important the tightness of these constraints is to you, and on the size of the portfolio relative to the prices of the assets.

The allowance computation is unlikely to be what is desired for `net.value`, so it is recommended that `net.value` always have length two. Section 6.4 on page 86 gives more details about constraining the value of the portfolio.

There are two cases that are of particular interest: dollar neutral portfolios and portfolios in the genre of 120/20.

Dollar Neutral Portfolios

A dollar neutral portfolio implies that the net value is zero. This is a case where the rule that `net.value` should only be given as an interval *might* be relaxed. The argument:

```
net.value = 0
```

translates into an interval that is symmetric around zero and the radius of the interval is the gross value times one minus `allowance`.

Even so it is probably better to explicitly set your interval for the net value.

120/20 and 130/30 Portfolios

The simplest way to get these is to give constraints just like the stated aim:

```
long.value = 1.2 * NAV, short.value = 0.2 * NAV
```

or possibly a range can be given:

```
long.value = c(1.2, 1.25) * NAV,
short.value = c(0.2, 0.25) * NAV,
net.value = c(.999, 1.0) * NAV
```

3.3 Limits on Assets

This section discusses the arguments:

- `max.weight`
- `enforce.max.weight`
- `universe.trade`
- `lower.trade`
- `upper.trade`
- `risk.fraction`
- `rf.style`
- `rf.loc`

max.weight

One typical use of the `max.weight` argument is:

```
max.weight = 0.05
```

This limits the maximum weight of each asset to 5% of the gross value of the portfolio. The weight is the absolute value of the monetary value of the position divided by the gross value. (So even in long-short portfolios the absolute weights always sum to 1.)

If you have a constraint like this, then it probably makes more sense to use the `risk.fraction` argument (see page 33) than `max.weight`.

The other typical use is to give `max.weight` a named vector that states the maximum weight for each asset. Assets that are not named are not restricted.

Example

If you want to allow a few assets to have larger weights, then you can create a vector of all the pertinent assets. Suppose you want almost all assets to have weight no more than 5% and a few to have weight no more than 7.5%. Then you could create the appropriate vector by:

```
> maxw.spec <- rep(0.05, length=length(prices))
> names(maxw.spec) <- names(prices)
> maxw.spec[spec.assets] <- 0.075
```

The `maxw.spec` vector would then be given as the `max.weight` argument.

A related argument is the control argument `enforce.max.weight`. When this is `TRUE` (the default) and if there are positions in the existing portfolio that will break the maximum weight constraint, then forced trades are created to make them conform to the maximum weight.

caution

There are circumstances in which the `max.weight` argument does not guarantee that the maximum weight in the final portfolio is obeyed if the weight is too large in the existing portfolio. A warning may not be given (since `max.weight` merely limits the extent of trading of the assets).

One case is if the control argument `enforce.max.weight` is `FALSE`. When this argument is `TRUE` (the default), then forced trades are automatically built to make the positions conform to their maximum weights. This is done for the maximum of the range of the gross value. If the range for the gross value is large and the resulting portfolio has a gross value substantially smaller than the maximum allowed gross, then some maximum weights could be broken.

It is also possible that a maximum weight is broken by enough that the trade can not be forced to be large enough. In this case, the trade is forced to be as large as possible if `enforce.max.weight` is `TRUE`.

If the maximum weight is the same for all assets, then you can ensure that it is obeyed by using `sum.weight` (well, either it's obeyed or you see a warning about it). To just constrain the largest weight, the `sum.weight` argument (see Section 3.14) would look like:

```
sum.weight = c("1"=.1)
```

This restricts the largest weight to 10%.

---

universe.trade

The `universe.trade` argument should be a vector of character strings containing some of the asset names. Those will be the only assets allowed to trade.

If this argument is `NULL` (the default), then there is no constraint on the assets to trade (except possibly for benchmarks).

lower.trade and upper.trade

The `lower.trade` and `upper.trade` arguments are given in asset units (shares, lots or whatever). As their names suggest, they limit the number of units that can be sold or bought. Like `max.weight` they can be given a single unnamed number or a vector with names corresponding to assets. Assets not named are not restricted.

Values in `lower.trade` can not be positive, and values in `upper.trade` can not be negative. This would be forcing the asset to trade. You can force trades, but not with these arguments—see `forced.trade` (page 38) or possibly `positions` (page 38).

One use of these arguments is to create a selection of assets to buy and a selection of assets to sell as seen on page 83 for long-short portfolios and on page 77 for long-only portfolios. More traditional uses include imposing liquidity constraints and constraining the amount of particular assets in the final portfolio.

Liquidity Constraints

Suppose that you want to constrain trades to a certain fraction of average daily volume. The commands you would use to do this would be similar to:

```
> liquidity <- 0.25 * ave.daily.volume
```

Then you could use this like:

```
upper.trade = liquidity, lower.trade = -liquidity
```

The process of constraining the liquidity merely involves computing the maximum amount of each asset that should be traded; setting the `upper.trade` argument to this amount; and setting the `lower.trade` argument to the negative of that amount.

In this example it is assumed that `ave.daily.volume` is a vector of numbers that has names which are the names of the assets—similar to the `prices` vector. These two vectors need not have the same assets. If there are assets in the daily volume vector (and hence in `liquidity`) that are not in `prices`, then these will be ignored. If `prices` has assets that are not in `liquidity`, then the missing assets will not be limited by the `upper.trade` and `lower.trade` arguments.

caution

Make sure that the units for the volume are the same as those for the prices. An easy mistake would be to try to limit trading to 10% of daily volume, but instead limit it to 10 times daily volume because the volume is in shares while the prices are in lots.

Another sort of liquidity constraint is to limit the amount of assets in the portfolio to n days of average volume. This is a case where using `positions` is an easier and safer approach. See page 39 for an example of getting this constraint using `positions`.

risk.fraction

The `risk.fraction` argument does what the `max.weight` argument is often unconsciously thought to do.

The variance of the portfolio can be partitioned into pieces attributed to each asset. In S notation with `w` denoting the weight vector and `V` the variance matrix, the partition is:

```
w * V %*% w  # equation rf1
```

and we can turn that into the fraction of variance with:

```
w * V %*% w / (w %*% V %*% w)  # equation rf2
```

By default the `risk.fraction` argument—if given—constrains the fractions of the variance in the partition. A command that includes:

```
risk.fraction = .05
```

means that all assets are constrained to have a variance fraction of no more than 5%.

Constraints need not be the same on all assets. An example of different constraints for different assets is:

```
risk.fraction = c(ABC=.1, DEF=.015, GHI=.07)
```

In the example above other assets would not have their variance fraction constrained. If these were the only assets, then it would be an error because the bounds sum to less than 1.

You can have lower constraints as well as upper constraints by using a two-column matrix. The first column is the lower bounds, and the upper bounds are in the second column. The assets are identified with the row names. Just as with a vector, not all assets need to be represented.

Benchmark

When there is a benchmark, then the equation above is slightly more complex:

```
(w - b) * V %*% (w - b) / ((w - b) %*% V %*% (w - b)) #eq rf3
```

where `b` is the vector of weights for the benchmark.

When there is a benchmark and you are constraining the variance fractions, you need to specify the benchmark weights (in terms of the other assets). Thus besides using the `risk.fraction` argument, you would do something like:

```
benchmark="Ben", bench.weights=list(Ben=c(A=.3, B=.3, C=.4))
```

Notice that the `bench.weights` argument takes a list with the name of each component being the name used for a benchmark.

If there is more than one variance-benchmark combination, then a three-dimensional array can be given as `risk.fraction` to specify the constraint bounds.

rf.style

What has been discussed so far in terms of risk fractions uses only the default value (`"fraction"`) for the `rf.style` argument. Table 3.2 lists the possible choices.

The `"fraction"` choice means that it is equation rf2 or rf3 that describes the quantities being constrained.

For `"value"` it is equation rf1 or its equivalent with a benchmark that will be used.

Marginal contribution to the benchmark

There is an argument to constrain the risk adjusted for the marginal contribution to the benchmark.

The marginal contribution is a number that can be computed as:

Table 3.2: Choices for the `rf.style` argument.

value	meaning
`"fraction"`	fraction of the portfolio variance for each asset
`"value"`	value of the portfolio variance for each asset
`"marginalbench"`	fraction of variance adjusted by marginal contribution
`"valmargbench"`	value of variance adjusted by marginal contribution
`"corport"`	correlation of assets with the portfolio
`"abscorport"`	absolute value of correlation with the portfolio
`"incorport"`	correlation for assets only in the portfolio
`"absincorport"`	absolute correlation for assets only in portfolio

```
b %*% V %*% (w - b)
```

The quantity that is being constrained with the `"valmargbench"` choice is:

```
(w - b) * (V %*% (w - b) - b %*% V %*% (w - b))  #eq rf4
```

The quantity for `"marginalbench"` is equation rf4 divided by the portfolio variance.

Correlation of assets to the portfolio

The correlation of each asset to the portfolio is sometimes of interest – usually related to the concept of dispersion.

One way of thinking about correlation is to start with equation rf1 and then scale it by the volatilities of the asset and the portfolio (instead of scaling by the portfolio variance).

A constraint like:

```
risk.fraction = .7
```

has various meanings depending on the value of `rf.style:`

- For style `"corport"` it means that all assets are to have a correlation with the portfolio less than 70% (a correlation of -80% is allowed).
- For style `"abscorport"` the absolute value of the correlation of all assets is to be less than 70% (-80% not allowed).
- For styles `"incorport"` and `"absincorport"` the constraints only apply to assets that are in the portfolio—an asset may have a correlation of 80% if it is not in the portfolio.

rf.loc

The `rf.loc` argument only applies if there is more than one variance-benchmark combination. It is the zero-based column number of `vtable` that is desired for each risk fraction constraint.

3.4 Number of Assets

This section discusses the arguments:

- `ntrade`
- `port.size`

Often these two constraints should be thought of as convenient shortcuts to threshold constraints (page 37).

Number of Assets to Trade

The `ntrade` argument controls the number of assets that may be traded. Generally it is given as a single number, meaning the maximum number to trade. The argument:

```
ntrade = 25
```

says that no more than 25 assets will be traded.

If you do not want a limit on the number of assets traded, then set `ntrade` to the size of the universe (or larger). The default is to have no limit.

If you desire a minimum number of assets traded, then give ntrade a length two vector. For example:

```
ntrade=c(4, 25)
```

states that the number of assets traded needs to be between 4 and 25, inclusive. A minimum number to trade is most likely to be useful when threshold constraints (Section 3.5) are used as well. Otherwise trading just one unit of an asset counts.

The minimum number to trade is more useful for generating random portfolios than in optimizing.

Number of Assets in the Portfolio

The `port.size` argument constrains the number of assets in the constructed portfolio. Like `ntrade`, this can be given as either a single number (meaning the upper bound), or as a length 2 vector giving the range.

The argument:

```
port.size = 50
```

means no more than 50 assets may be in the portfolio, while:

```
port.size = c(50, 50)
```

means that exactly 50 assets need to be in the portfolio.

3.5 Threshold Constraints

This section discusses the argument:

- `threshold`

The `threshold` argument controls two types of threshold constraints—trade thresholds and portfolio thresholds. Threshold constraints may also be specified with `positions` (page 38). Note that in any one call you can only declare threshold constraints with one of these arguments (though both arguments can be used in one call).

Trade Thresholds

Trade threshold constraints force trades to be at least a certain amount of an asset if it is traded at all.

The argument:

```
threshold=c(ABC=4, DEF=23))
```

demands that at least 4 units (lots perhaps) of asset ABC and at least 23 units of DEF be bought or sold if they are traded at all. This is not the same as forcing a trade of a certain size—that is discussed in Section 3.6.

When the `threshold` argument is a vector (or a one-column matrix), then the constraint is taken to be symmetric. Another way of stating the constraint given above would be:

```
threshold=rbind(ABC=c(-4,4), DEF=c(-23,23)))
```

where what we are giving to `threshold` looks like:

```
> rbind(ABC=c(-4,4), DEF=c(-23,23)))
    [,1] [,2]
ABC   -4    4
DEF  -23   23
```

Give a two-column matrix when you want different constraints for buying and selling for at least one of the assets.

```
threshold=rbind(ABC=c(-5,4), DEF=c(0,23)))
```

The command above states that if ABC is sold, then at least 5 units should be sold. If ABC is bought, then at least 4 units should be bought. If DEF is bought, then at least 23 units should be bought. There is no threshold constraint if DEF is sold.

Portfolio Thresholds

A portfolio threshold constraint states the minimum number of units of an asset that should be held in the portfolio. For example, you may prefer to have no lots of ABC if less than 7 lots are held. Portfolio thresholds are specified similarly to trade thresholds except they are in the third and fourth columns of the matrix instead of the first and second.

Here is an example:

```
> thresh.m1 <- cbind(0, 0, rbind(ABC=c(-7, 7), DEF=c(-5, 8)))
> thresh.m1
    [,1] [,2] [,3] [,4]
ABC    0    0   -7    7
DEF    0    0   -5    8
```

If `thresh.ml` is given as `threshold`, then there should be at least 7 units of ABC—either long or short—if it is in the portfolio at all. Also DEF should be at least 5 short or at least 8 long or not in the portfolio. In this case there are no trading threshold constraints since the first two columns are all zero.

Summary of Threshold Inputs

- A vector or a one-column matrix: symmetric trading constraints.
- A two-column matrix: symmetric or asymmetric trading constraints.
- A three-column matrix: first two columns are trading constraints, third column is portfolio constraint for long-only portfolios. It is an error to give a three-column matrix with long-short portfolios.
- A four-column matrix: first two columns are trading constraints, third and fourth columns are portfolio constraints.

3.6 Forced Trades

This section discusses the argument:

- `forced.trade`

The `forced.trade` argument is a named vector giving one or more trades that must be performed. The value gives the minimal amount to trade and the names give the assets to be traded. For example:

```
forced.trade = c(ABC=-8, DEF=15)
```

says to sell at least 8 units (lots) of ABC and buy at least 15 units of DEF. If you wanted to buy exactly 15 units of DEF, then you would say:

```
forced.trade = c(ABC=-8, DEF=15), upper.trade=c(DEF=15)
```

Trades may also be automatically forced if the existing portfolio breaks maximum weight constraints—see page 32). The `positions` argument can also force trades.

3.7 Positions

This section discusses the arguments:

- `positions`

- `tol.positions`

The `positions` argument does not embody any constraints that can't be achieved with other arguments. It exists because the constraints can sometimes be more conveniently expressed via `positions`.

The constraints that positions can do are:

- `max.weight`
- `universe.trade`
- `lower.trade`
- `upper.trade`
- `threshold`
- `forced.trade`

The key difference between `positions` and these other constraints is that `positions` is expressed in monetary value. In examples we will assume the currency unit is dollars, but it is really the currency unit that the `prices` vector is in.

The `positions` argument takes a matrix with rows representing assets, and 2, 4 or 8 columns. Not all assets in the problem need to be present in `positions` but (by default) there will be a warning if not.

Portfolio constraints

The first two columns of `positions` contain portfolio constraints. Column 1 is the minimum amount of money that is allowed in the portfolio for each of the assets. Column 2 is the maximum amount of money.

So this includes the `max.weight` constraint. If both `positions` and `max.weight` are given, then whichever is the stronger for each asset is the actual constraint.

n days of daily volume in portfolio

On page 33 there is an example of imposing liquidity constraints on the trade. Here we want to impose liquidity constraints on the portfolio.

Suppose that we want to restrict positions in the portfolio to be no more than 8 days of average volume. Since `positions` is expecting monetary value rather than the number of asset units, we first need to transform to money:

```
> liquid.value <- ave.daily.volume[names(prices)] * prices
> any(is.na(liquid.value)) # trouble if this is TRUE
```

There are a couple things to note here. We are assuming that the volume and the prices refer to the same asset units—we don't want one to be in shares and the other in lots. We'll see on page 41 why the occurrence of missing values would be upsetting.

Now we are ready to build our matrix to give to the `positions` argument. If we have a long-only portfolio, then the matrix can be:

```
> posmat1 <- cbind(0, 8 * liquid.value)
```

If we have a long-short portfolio, then it would be:

```
> posmat2 <- 8 * cbind(-liquid.value, liquid.value)
```

Let's do one more supposition. Suppose that we have a long-only portfolio and we want no position to be larger than $300,000 as well as having the liquidity constraint. You might be tempted to impose that with a command like:

```
> posmat1[, 2] <- min(3e5, posmat1[, 2]) # WRONG
```

Note that there is no warning from this—S has no way of knowing that you are doing something you don't really want to do. What you do want to do is:

```
> posmat1[, 2] <- pmin(3e5, posmat1[, 2]) # right
```

The `min` function returns just one number which is the minimum of all the numbers it sees. The `pmin` function does a minimum for each element, which is what we want in this instance.

Trade constraints

Trade constraints are imposed with the third and fourth columns of `positions`. If you want to impose trade constraints without imposing portfolio constraints, then you can just make the first two columns infinite. For example, if we want selling of each asset to be limited to $5000 and buying of each asset to be limited to $8000, then we could create a matrix like:

```
> posmat3 <- cbind(rep(-Inf, length(prices)), Inf,
+    -5000, 8000)
> head(posmat3)
     [,1] [,2]  [,3] [,4]
[1,] -Inf  Inf -5000 8000
[2,] -Inf  Inf -5000 8000
[3,] -Inf  Inf -5000 8000
[4,] -Inf  Inf -5000 8000
[5,] -Inf  Inf -5000 8000
[6,] -Inf  Inf -5000 8000
> dimnames(posmat3) <- list(names(prices), NULL)
```

These constraints are essentially identical to `lower.trade` and `upper.trade` except they are expressed in money rather than asset units, and it is possible to impose forced trades with the `positions` constraints.

Forced constraints

A trade is forced either if the range for the portfolio does not include the current position, or if the trading range does not include zero. This substitutes for using `forced.trade`.

Forced trades from `positions` cause the output of `trade.optimizer` to have a component named `positions.forced` which will show the forced trades created.

Universe constraints

If there is a missing value (NA) in any of the first four columns of positions, then the corresponding asset is not allowed to trade. (It is an error to have a missing value in the fifth through eighth columns.) These constraints could be achieved by using the universe.trade argument.

Be careful not to let missing values stray in by accident. The summary of optimized portfolios tells how many assets positions forces not to trade.

Threshold constraints

The 5th through the 8th columns of positions are for threshold constraints. All four of these columns need to be given if any are given. Note also that if these columns are given, then the threshold argument can not be given.

You can put null constraints in columns that you don't want to have influence. The most extreme possibility in this regard is if you want to use positions to only impose a portfolio threshold on a long-only portfolio. Suppose you want only positions that have at least $20,000, then you could build the positions matrix like:

```
> posmat4 <- cbind(rep(-Inf, length(prices)), Inf, -Inf, Inf,
+     0, 0, 0, 2e4)
> head(posmat4)
     [,1] [,2] [,3] [,4] [,5] [,6] [,7]  [,8]
[1,] -Inf  Inf -Inf  Inf    0    0    0 20000
[2,] -Inf  Inf -Inf  Inf    0    0    0 20000
[3,] -Inf  Inf -Inf  Inf    0    0    0 20000
[4,] -Inf  Inf -Inf  Inf    0    0    0 20000
[5,] -Inf  Inf -Inf  Inf    0    0    0 20000
[6,] -Inf  Inf -Inf  Inf    0    0    0 20000
> dimnames(posmat4) <- list(names(prices), NULL)
```

Tolerance

Imagine a situation where asset ABC is restricted to be no more than $50,000 in the portfolio. At time 1 the optimizer likes ABC and buys enough that it is close to that upper limit. From time 1 to time 2 ABC dutifully does well, so now it is above the $50,000 limit. Assuming the limit remains the same for time 2, then the optimizer will force a sale of some ABC. So we are performing a trade (probably of a trivial amount) for a rather arbitrary reason.

If there is a trade threshold constraint on selling ABC, then it is even worse—we are effectively forcing a sale of at least the threshold amount.

The tol.positions argument solves this problem. If tol.positions is zero (the default), then the constraints are taken as stated. If tol.positions is positive, then assets that violate portfolio constraints in positions by a value less than tol.positions are not forced to trade.

In our example if the value of the position of ABC at time 2 is $50,203, then at least $203 worth of ABC will be forced to be sold if tol.positions is less than $203. But there will not be a forced trade if tol.positions is greater than $203.

Note that this tolerance only applies to assets that are already in the existing portfolio.

Summary of positions inputs

- 2-column matrix: lower and upper constraints on the value of portfolio positions.
- 4-column matrix: 1-2) constraints on portfolio positions; 3-4) constraints on the value of sells and buys in the trade.
- 8-column matrix: 1-2) constraints on the portfolio positions; 3-4) constraints on the trade; 5-6) threshold constraints on the values of trades; 7-8) threshold constraints on the values in the portfolio. (Note: portfolio-trade-trade-portfolio.)

3.8 Linear Constraints

This section describes the arguments:

- `lin.constraints`
- `lin.bounds`
- `lin.trade`
- `lin.abs`
- `lin.style`
- `lin.direction`
- `lin.rfloc`

These arguments create linear constraints on the portfolio and/or the trade. These arguments can also impose count constraints—a topic that is discussed in the next section (page 52).

Portfolio Probe makes a distinction between numeric constraints and constraints based on categories. An example of the first is a bound on twice the value of asset `ABC` minus 1.5 times the value of asset `DEF` minus the value of asset `GHI`. An example of the second is bounds on the value from each country in the asset universe.

Linear constraints are used when there is aggregation across assets. If a constraint only involves one asset at a time, then there are other constraints that are more expedient—for instance, `max.weight, risk.fraction` or `threshold`.

Building Constraints

To impose linear constraints you must specify at least two arguments—you can accept default values of other arguments. The two key arguments are:

- `lin.constraints`

- lin.bounds

The lin.constraints argument gives the numbers or categories for each asset for each constraint. lin.bounds provides the lower and upper bound for each (sub)constraint. Because the functions are picky about these arguments, it is best to create them with the build.constraints function.

The build.constraints function takes a vector, a matrix, a factor or a data frame containing the information for the lin.constraints argument. The vector or matrix can be character, numeric or logical. A data frame is required if you have a mixture of types.

In this example, we are giving a character matrix with two columns—one for countries and one for sectors:

```
> cons.obj <- build.constraints(cbind(country=countryvec,
+     sector=sectvec))
> names(cons.obj)
[1] "lin.constraints" "bounds"
```

The result of build.constraints is a list with two components. The first component, lin.constraints, is suitable to give as the lin.constraints argument, the bounds component is the template for an object to give as lin.bounds. The lin.constraints object must have column names, so build.constraints will add column names if they are not already there. (The column names are used to keep track of which bounds go with which constraint.)

```
> cons.obj$lin.constraints[1:3,]
     country   sector
ABC "Spain"  "media"
DEF "France" "retail"
GHI "Italy"  "energy"
> cons.obj$bounds
                 lower upper
country : France  -Inf   Inf
country : Italy   -Inf   Inf
country : Spain   -Inf   Inf
sector : energy   -Inf   Inf
sector : media    -Inf   Inf
sector : retail   -Inf   Inf
sector : telecom  -Inf   Inf
```

The typical use of the bounds component is to create a separate object out of it, and then change any of the infinite values desired—there are examples below.

Bounds and lin.style

What the bounds look like depends on the style of the constraint. Table 3.3 shows the possible values that can be given for lin.style.

For historical reasons the default for lin.style is "weight". However, from an investment point of view, it might be better if the default were "varfraction".

Table 3.3: Choices for the `lin.style` argument.

value	**meaning**
`"weight"`	aggregate of weights (position value divided by gross)
`"value"`	aggregate of monetary value
`"varfraction"`	aggregate of asset fraction of variance
`"varvalue"`	aggregate of asset contribution to variance
`"varmbfraction"`	variance fraction adjust for marginal contrib to benchmark
`"varmbvalue"`	variance value adjust for marginal contrib to benchmark
`"count"`	number of assets (see Section 3.9)

The variance fraction constraint is almost surely closer to what is in the investor's head than the weight constraint.

If we want bounds in terms of weights (or a variance fraction), then we could create the bounds matrix something like:

```
> bounds.csw <- cons.obj$bounds
> bounds.csw[1:2, ] <- c(.1, .2, .4, .5)
> bounds.csw[3, 2] <- .2
> bounds.csw[5, ] <- c(.3, .6)
> bounds.csw
                  lower upper
country : France    0.1   0.4
country : Italy     0.2   0.5
country : Spain    -Inf   0.2
sector : energy    -Inf   Inf
sector : media      0.3   0.6
sector : retail    -Inf   Inf
sector : telecom   -Inf   Inf
```

We could then use the arguments:

```
lin.constraints = cons.obj$lin.constraints,
lin.bounds = bounds.csw, gross.value = 1e6
```

If on the other hand, we want bounds in terms of monetary value, then we could do:

```
> bounds.csv <- cons.obj$bounds
> bounds.csv[1:2, ] <- c(1e5, 2e5, 4e5, 5e5)
> bounds.csv[3, 2] <- 2e5
> bounds.csv[5,] <- c(3e5, 6e5)
> bounds.csv
                  lower upper
country : France 1e+05 4e+05
country : Italy  2e+05 5e+05
country : Spain   -Inf 2e+05
sector : energy   -Inf   Inf
sector : media   3e+05 6e+05
sector : retail   -Inf   Inf
sector : telecom  -Inf   Inf
```

Now the arguments would be:

```
lin.constraints = cons.obj$lin.constraints,
lin.bounds = bounds.csv, gross.value = 1e6,
lin.style = "value"
```

In this case we need to give the `lin.style` argument because we are using a non-default value for it.

It is also possible to mix styles:

```
> bounds.csmix <- bounds.csw
> bounds.csmix[1:3, ] <- bounds.csv[1:3, ]
> bounds.csmix
                 lower upper
country : France 1e+05 4e+05
country : Italy  2e+05 5e+05
country : Spain   -Inf 2e+05
sector : energy   -Inf   Inf
sector : media     0.3   0.6
sector : retail   -Inf   Inf
sector : telecom  -Inf   Inf
```

Changing the problem slightly, the arguments become:

```
lin.constraints = cons.obj$lin.constraints,
lin.bounds = bounds.csv, gross.value = 1e6,
lin.style = c("value", "varfraction")
```

There is no need for any particular pattern of finite bounds. In this example, the only sector we are bounding is media.

Once a bounds matrix has been set up, it can be used when building new constraint objects. Suppose that we want to change from sectors to industries, we can build new constraints like:

```
> cons.obj2 <- build.constraints(cbind(country=countryvec,
+     industry=industvec), bound=bounds.csw)
```

S code note

In the `cbind` command, we are assuming that `countryvec` and `industvec` have the same assets in the same order. A safer approach would be:

```
> ci.inam <- intersect(names(countryvec), names(industvec))
> cbind(country=countryvec[ci.inam],
+     industry=industvec[ci.inam])
```

The `bounds` component of `cons.obj2` will have the same bounds for the countries as `bounds.csw` and will have infinite bounds for the industries.

A bounds object that is actually used may contain extraneous bounds—for example bounds for sectors when only countries are being constrained. However, it is an error not to give bounds for all of the constraints represented in `lin.constraints`.

Linear Constraints on Variance Partitions

Styles `"varvalue"` and `"varmbvalue"` are really quadratic constraints, and styles `"varfraction"` and `"varmbfraction"` are dynamic quadratic constraints. But given the partitions of variance (see the discussion of `risk.fraction` in Section 3.3), then the constraints done here are just linear aggregations.

When an investor wants a constraint of 20% to 30% on the Energy sector, they very likely want to constrain the fraction of risk to that range rather than the fraction of money. The technology for the former didn't used to be available—hence the ubiquity of the latter.

There are some restrictions on the variance partition constraints:

- `lin.abs` must be `TRUE` for the variance partition constraints, `FALSE` values are coerced to be `TRUE`. (Switching signs of variance partitions and then summing doesn't make sense, which is the image that at least some would have.)
- `lin.trade` must be `FALSE`—the variance for the trade is not computed.

If there is more than one variance-benchmark combination, then the `lin.rfloc` argument specifies which column of `vtable` (see page on page 126) is to be used with each constraint. It is zero-based, and ignored for constraints that don't use the variance.

Numerical Constraints: Risk Factors

Only categorical constraints have been discussed so far. We now look at numerical constraints.

Constraining risk factors is perhaps the most common numerical linear constraint. In a fixed income portfolio duration would be another example.

Whether constraints are numerical or categorical, the first step is to use `build.constraints`:

```
> cons.obj3 <- build.constraints(cbind(Risk1=beta1))
> head(cons.obj3$lin.constraints)
           Risk1
stockA 1.3119461
stockB 0.9886379
stockC 1.1688637
stockD 0.8160228
stockE 1.0312180
stockF 1.2067453
> cons.obj3$bounds
```

```
      lower upper
Risk1  -Inf   Inf
> cons.obj3$bounds[] <- c(0.9, 0.95)
> cons.obj3$bounds
      lower upper
Risk1   0.9  0.95
```

S code note

Two things:

First, the use of `cbind` in the first line of the example above is merely an easy way to create a one-column matrix that has a name for the column.

Second, you might be tempted to do:

```
> cons.obj3$bounds <- c(0.9, 0.95)
```

instead of

```
> cons.obj3$bounds[] <- c(0.9, 0.95)
```

The difference is that the first of these will result in a plain vector while the second retains the matrix structure of the original object. We want that matrix structure—the row names in particular.

We would now use the arguments:

```
lin.constraints = cons.obj3$lin.constraints,
lin.bounds = cons.obj3$bounds
```

Constraining more than one risk factor is straightforward.

```
> cons.obj4 <- build.constraints(cbind(Risk1=beta1,
+    Risk2=beta2))
> head(cons.obj4$lin.constraints)
           Risk1     Risk2
stockA 1.3119461 0.8972907
stockB 0.9886379 0.9784834
stockC 1.1688637 0.7135961
stockD 0.8160228 1.1195767
stockE 1.0312180 1.0847337
stockF 1.2067453 1.0977745
```

Now just create the bounds that you want and use the arguments as before.

Numerical Constraints: Market Capitalization

If you want to have a portfolio that has a similar average market capitalization as an index, then you can impose this as a linear numerical constraint. We assume here that the index is weighted by market capitalization and we have the weights for the index.

The first step is to use `build.constraints` as before:

```
> cons.mcap <- build.constraints(cbind(Mcap=index.wts))
```

The possibly surprising part is what the bound looks like. The value for this constraint (assuming a weight style as opposed to value) that the index has is the sum the squared weights:

```
> mcap.target <- sum(index.wts^2)
```

The bounds you use might, for example, be 95% and 105% of the target.

Mixing Numerical and Categorical Constraints

There is one tricky part when you have both numerical and categorical linear constraints. Up until now we have used a matrix (or a vector) as the first argument to `build.constraints`. A matrix has to have all of its elements of the same type, but now we want a mix of types: numeric for the numerical constraints, and character, logical or factor for the categorical constraints. Now we need to use a data frame, not a matrix, to represent our constraint data.

```
> cons.objmix <- build.constraints(data.frame(Risk1=beta1,
+    Risk2=beta2, Country=country))
> head(cons.objmix$lin.constraints)
           Risk1     Risk2   Country
stockA 1.3119461 0.8972907 Australia
stockB 0.9886379 0.9784834 Singapore
stockC 1.1688637 0.7135961 Japan
stockD 0.8160228 1.1195767 Australia
stockE 1.0312180 1.0847337 Japan
stockF 1.2067453 1.0977745 Australia
> cons.objmix$bounds
                   lower upper
Risk1               -Inf   Inf
Risk2               -Inf   Inf
Country : Australia -Inf   Inf
Country : Japan     -Inf   Inf
Country : Singapore -Inf   Inf
```

Yet again the next step is to specify the bounds and then use the arguments.

Portfolio Constraints versus Trade Constraints

The default behavior is to constrain the portfolio. If you want the trade to be constrained rather than the portfolio, then set the `lin.trade` argument to `TRUE`. You can have a mixture of constraints on the trade and on the portfolio. The vector given as `lin.trade` is replicated to have length equal to the number of columns in the `lin.constraint` object.

If `lin.constraints` has three columns, then the argument:

```
lin.trade = c(TRUE, FALSE, TRUE)
```

means that the constraint in the first column is on the trade, the second column constrains the portfolio and the third column constrains the trade.

Net Constraints versus Gross Constraints

By default, constraints use the sum of the absolute value of weights or value or counts. If you want the constraints on the net rather than the gross, then set the `lin.abs` argument to be `FALSE`. For the variance partition styles, `lin.abs` must be `TRUE` (the default value switched in version 1.04 because of this).

For example, in a long-short portfolio you may want to limit the net weight in a sector, but you may also want to limit the gross amount in the sector.

The `lin.abs` argument is replicated to have length equal to the number of columns in the `lin.constraint` object.

As with similar arguments, `lin.abs` can be a vector:

```
lin.abs = c(TRUE, TRUE, FALSE)
```

Let's look at the four possible combinations of the values of `lin.trade` and `lin.abs`. These are:

- portfolio, gross (absolute) (the default)
- portfolio, net (not absolute)
- trade, gross (absolute)
- trade, net (not absolute)

For long-only portfolios the first two are the same since all positions are positive.

An extreme example is to enforce all four types on the same constraint. You want to name the constraints properly so that you can keep track of them:

```
> con.4c <- build.constraints(cbind(c.tn=countryvec,
+    c.ta=countryvec, c.pn=countryvec, c.pa=countryvec))
```

The result of the above command can then be used like:

```
lin.constraints = con.4c$lin.constraints,
lin.trade = c(TRUE,TRUE,FALSE,FALSE),
lin.abs = c(FALSE,TRUE))
```

The `lin.abs` argument will automatically be replicated to be: FALSE, TRUE, FALSE, TRUE. Of course we could have given all four values.

Long-side Constraints and Short-side Constraints

This section is mainly for long-short portfolios, in which it applies to both trade constraints and portfolio constraints.

For long-only portfolios it can apply to trade constraints. In general, it does not apply to portfolio constraints, but there is a catch with variance partition constraints as described below.

A long-side constraint only involves assets that have long positions—the short positions are effectively treated as zero for the purposes of the constraint. Likewise, a short-side constraint only looks at assets with short positions.

The `lin.direction` argument controls whether you have a long-side, short-side or both-side constraint.

There are three possible values:

- `lin.direction = 0` means both-sides (the default)
- `lin.direction = 1` means long-side
- `lin.direction = -1` means short-side

Of course, `lin.direction` gets replicated to be as long as the number of columns in the `lin.constraints` object, and can be given as a vector:

```
lin.direction = c(1, -1, 0, 0)
```

When a short-side constraint is in effect (`lin.direction=-1`), then it is the absolute value of the short values that is used—your constraints should be non-negative.

With the weight, value and count styles an asset not in the portfolio (or trade) is always zero. However, assets not in the portfolio can have a non-zero contribution to the variance. So a variance partition constraint can be different for a long-only portfolio when `lin.direction` is 1 rather than 0.

Looking at the Effect of the Constraints

The `constraints.realized` function displays the status of linear constraints for a portfolio. This function produces the `con.realized` component of objects returned by `trade.optimizer`. This component is not a part of the printing of the object, but is a part of the summary. (Using `summary` can give you information on some of the other constraints as well.) Here is an example:

```
> opti2$con.realized
$linear
                lower  upper realized nearest violation
c.tn : France -3e+05 -2e+05  -296632   -3368        NA
c.tn : Italy   1e+05  2e+05   180767   19233        NA
c.tn : Spain  -1e+05 -3e+04   -75494  -24506        NA
c.ta : France  6e+05  7e+05   600624    -624        NA
c.ta : Italy   2e+05  3e+05   206917   -6917        NA
c.ta : Spain   4e+05  5e+05   399552      NA      -448
c.pn : France -1e+05  0e+00   -24099   24099        NA
c.pn : Italy   0e+00  1e+05    92422    7578        NA
c.pn : Spain  -1e+05  0e+00   -76340  -23660        NA
c.pa : France  3e+05  4e+05   394621    5379        NA
c.pa : Italy   3e+05  4e+05   304686   -4686        NA
c.pa : Spain   3e+05  4e+05   400398      NA       398
```

This contains a matrix where each row has information on a sub-constraint. The final column indicates the size of violations—if all of the elements in this column are `NA`, then there are no violations. A negative violation means that the realized value is below the lower bound, and a positive violation means that the realized value is above the upper bound. The next to last column gives proximity of the realized value to the nearest bound (if there is not a violation). The same convention in terms of sign is used—a negative number means the realized is closer to the lower bound. The last two columns have exactly one missing value for each row.

In this example the sizes of the violations are relatively trivial. However, they would not be trivial in terms of their affect on the objective if you were optimizing—any violation means that the optimizer has been concentrating on getting a feasible solution (a solution that merely satisfies the constraints) and may not be near the best feasible solution. At times there is a non-zero penalty that is of trivial size relative to the utility, you need not worry in such cases.

Evaluating Un-imposed Constraints

It is possible to look at the value of constraints that were not imposed on a portfolio. Give `constraints.realized` the portfolio object (the result of a call to `trade.optimizer`) and the constraints (as a matrix or data frame) that are of interest:

```
> constraints.realized(opt.ir2, cbind(country=countryvec))
                 lower upper  realized nearest violation
country : France  -Inf   Inf  0.020106    -Inf        NA
country : Italy   -Inf   Inf  0.156728    -Inf        NA
country : Spain   -Inf   Inf -0.157020    -Inf        NA
```

You may also use the additional linear constraint arguments like `lin.abs` and `lin.trade` to specify the nature of the constraints you want:

```
> constraints.realized(opt.ir2, cbind(country=countryvec),
+    lin.abs=TRUE, lin.style="value")
                 lower upper realized nearest violation
country : France  -Inf   Inf   433494    -Inf        NA
country : Italy   -Inf   Inf   296702    -Inf        NA
country : Spain   -Inf   Inf   669666    -Inf        NA
```

Inspecting Linear Constraints

The `summary` for portfolios includes the result of `constraints.realized` on the portfolio as well as listing the effective values of the arguments that control the type of constraints. If you are doing optimization, you can get this with:

```
opt.obj <- trade.optimizer( ... )
summary(opt.obj)
```

If you are generating random portfolios, you can do:

```
rp.obj <- random.portfolio( ... )
randport.eval(rp.obj, subset=1, FUN=summary)
```

Setting the subset argument to 1 means that we are only going to evaluate the first random portfolio.

An example for random portfolios is:

```
> tail(randport.eval(rp.obj6, FUN=summary,
+    subset=1)[[1]], 1)
$constraints.realized
$constraints.realized$linear
                  lower upper realized nearest violation
Country : France   -300     0   -28.09   28.09        NA
Country : Germany     0   100    27.57  -27.57        NA
Country : UK        100  2000   175.39  -75.39        NA
```

An example for optimization is:

```
> tail(summary(op.obj6), 1)
$constraints.realized
$constraints.realized$linear
                 lower   upper  realized    nearest violation
Cntry : France  -300.0    0.00 -297.6500 -2.350e+00        NA
Cntry : Germany    0.0  100.00   11.4600 -1.146e+01        NA
Cntry : UK       100.0 2000.00  782.1500 -6.822e+02        NA
Beta               0.9    0.95    0.9001 -1.060e-04        NA
```

3.9 Count Constraints

This section continues the description of arguments:

- lin.constraints
- lin.bounds
- lin.trade
- lin.abs
- lin.style
- lin.direction

but with the restriction that lin.style="count".

When lin.style is "count", then count constraints—user-defined integer constraints— are produced. This can only be used with a categorical column of lin.constraints—a count constraint on a numerical column will trigger an error.

Let's consider some examples.

Suppose that we have a long-short portfolio and we want an equal number of long positions and short positions from the UK, and we want one more French long position than French short position. We have no preference regarding Germany. Then we would create a bounds object like:

```
> lbc1
                    lower upper
Country : France      0.5   1.5
Country : Germany    -Inf   Inf
Country : UK         -0.5   0.5
```

Our call would then include the arguments:

```
lin.bounds = lbc1, lin.style = "count", lin.abs = FALSE
```

We can accept the default value of `lin.trade` because we want a constraint on the portfolio. We can infer from the bounds object that the `lin.constraints` argument contains just one constraint that is called `Country`.

Notice that in this context (`lin.abs = FALSE`, `lin.direction = 0`) long positions count as one but short positions count as negative one. Also notice that we are giving bounds that are between integers—you don't want to give bounds for count constraints that are integer.

Now suppose that we want exactly one German position in the portfolio and we want one or two French positions. Our bounds object should now look like:

```
> lbc2
                    lower upper
Country : France      0.5   2.5
Country : Germany     0.5   1.5
Country : UK         -Inf   Inf
```

We would have arguments:

```
lin.bounds = lbc2, lin.style = "count", lin.abs = TRUE
```

If we want exactly two long UK positions and we want no short French positions, then we need to do a bit more work. The extra work is because we now have two constraints (a long-side constraint and a short-side constraint) so we need two columns in `lin.constraints`, not the one column that we have been using. We start by building the appropriate object:

```
> lcc3 <- build.constraints(cbind(Country.longc=countryvec,
+   Country.shortc=countryvec))
```

Now we get the bounds matrix that we need:

```
> lbc3 <- lcc3$bounds
> # lbc3
> lbc3[3,] <- c(1.5, 2.5)
> lbc3[4, 2] <- .5
> lbc3
                           lower upper
Country.longc : France      -Inf   Inf
Country.longc : Germany     -Inf   Inf
Country.longc : UK           1.5   2.5
Country.shortc : France     -Inf   0.5
Country.shortc : Germany    -Inf   Inf
Country.shortc : UK         -Inf   Inf
```

Now we'll use arguments:

```
lin.constraints = lcc3$lin.constraints, lin.bounds = lbc3,
lin.style = "count", lin.direction = c(1, -1)
```

Logical values are allowed to define constraints. Logicals are probably most likely to be used in count constraints. Here we have a somewhat contrived example (as if the others aren't) where we want to lower the beta of the portfolio. We lower beta if we sell high beta stocks or buy low beta stocks. We'll create a constraint that does more of this than its opposite in terms of asset count. We build the constraint object and the bounds:

```
> lcc4 <- build.constraints(cbind(HighBeta = beta > 1)
> lbc4 <- lcc4$bounds
> lbc4[1,1] <- 3.5
> lbc4
                  lower upper
HighBeta : FALSE   3.5   Inf
HighBeta : TRUE   -Inf   Inf
```

Our arguments are:

```
lin.constraints = lcc4$lin.constraints, lin.bounds = lbc4,
lin.style = "count", lin.trade = TRUE
```

You might want to constrain the number of assets that are long and/or short in the portfolio. You can do that by creating a categorical variable with only one level. One approach is to use `TRUE` as the level.

```
> assets <- rep(TRUE, length(prices))
> names(assets) <- names(prices)
> assetcon1 <- build.constraints(cbind(assets=assets))
> assetcon1$bounds[] <- c(23.5, 30.5)
```

In the code above we have set up the constraints so that the count can be 24 to 30, inclusive (the bounds should be away from integers for count constraints). To impose a constraint on the number long, we would use this like:

```
lin.bounds=assetcon1$bounds, lin.direction=1
```

To constrain the number of positions both long and short, we could set it up like:

```
> assetcon2 <- build.constraints(cbind(nlong=assets,
+     nshort=assets))
> assetcon2$bounds[1,] <- c(23.5, 30.5)
> assetcon2$bounds[2,] <- c(19.5, 25.5)
```

and then use it like:

```
lin.bounds=assetcon2$bounds, lin.direction=c(1,-1)
```

3.10 Alpha (Expected Return) Constraints

This section describes the argument:

- `alpha.constraint`

The `alpha.constraint` argument bounds the expected return of the optimized portfolio. In its simplest use, it is merely a single number that gives the minimum for the expected return. For example:

```
alpha.constraint = 1.1
```

An upper bound as well as a lower bound can be specified by giving a two-column matrix. For instance:

```
alpha.constraint = cbind(1.1, 2.3)
```

will restrict the expected return to be between 1.1 and 2.3.

Note that if a benchmark is given, then this will constrain the expected return relative to the benchmark. This is just the usual expected return of the portfolio minus the expected return of the benchmark.

Advanced use of `alpha.constraint` (chiefly when more than one vector of expected returns is given) is discussed on page 129.

3.11 Variance Constraints

This section discusses the argument:

- `var.constraint`

The `var.constraint` argument constrains the value of the variance. In its most common usage, it is merely a number giving the upper bound for the variance. Using the argument:

```
var.constraint = 1.4
```

imposes an upper bound of 1.4 on the variance.

The next most common usage is a two column matrix that gives a range of values that the variance is allowed to be in. For example:

```
var.constraint = cbind(1.2, 1.4)
```

says that the variance is to be at least 1.2 but no more than 1.4. The value given looks like:

```
> cbind(1.2, 1.4)
     [,1] [,2]
[1,]  1.2  1.4
```

More advanced use of this argument—predominantly when multiple variances are given—is discussed on page 127.

3.12 Tracking Error (Benchmark) Constraints

This section discusses the argument:

- `bench.constraint`

The `bench.constraint` argument is used to constrain (squared) tracking errors. There has to be information on not only the numerical bound that you want, but the benchmark as well. The benchmark needs to be an asset in the variance, but it doesn't need to be in the prices unless benchmark trading is allowed.

Ultimately `bench.constraint` creates a variance constraint—it is just a more convenient way of specifying some commonly desired constraints.

Single Upper Bound

If you want just an upper bound, then give a named numeric vector of length one. For example:

```
bench.constraint = c(spx = .04^2)
```

specifies a 4% tracking error from `spx`.

Another way of doing the same thing is:

```
> bc <- .04 ^ 2
> names(bc) <- "spx"
> bc
   spx
0.0016
```

and then use the argument:

```
bench.constraint = bc
```

Scaling

Don't forget that this constraint is on squared tracking error, not tracking error.

While tracking errors are almost always an annual number, your data probably aren't annual. For example if your variance is on a daily scale, then you need to transform the constraint to the daily scale:

```
bench.constraint = c(spx = .04^2 / 252)
```

Lower and Upper Bounds

Lower bounds as well as upper bounds can be specified by giving a two-column matrix. For example:

```
bench.constraint = rbind(spx=c(.02, .04)^2)
```

would restrict `spx` tracking error to between 2% and 4%.

This is most likely of interest when generating random portfolios.

Multiple Benchmarks

If you want upper bounds on more than one benchmark, merely give a numeric vector with the names of the benchmarks:

```
bench.constraint = c(spx = .04^2, R1000 = .05^2)
```

specifies no more than a 4% tracking error from `spx` and also no more than a 5% tracking error from `R1000`.

To get both upper and lower constraints for multiple benchmarks, you give a two-column matrix with multiple rows. For example:

```
bench.constraint = mult.benchcon.lowup
```

where

```
> mult.benchcon.lowup
        [,1]   [,2]
spx   0.0004 0.0016
R1000 0.0009 0.0025
R2000 0.0016 0.0064
```

Advanced Use

When there are multiple variance matrices, then specifying benchmark constraints is more complicated. This is discussed on page 127.

3.13 Distance

This section discusses the arguments:

- `dist.center`
- `dist.style`
- `dist.bounds`
- `dist.trade`
- `dist.utility`
- `dist.prices`

These arguments impose distance constraints.

Let's start with a simple example. Suppose we have a target portfolio that has the following weights:

```
> targwt <- c(Asset1=.4, Asset2=.3, Asset3=.2, Asset4=.1)
```

Suppose we want the resulting portfolio to have weight distance no more than 0.5 (so the buys plus sells of the trade to get from the result to the target would be no more than 50% of the gross value of the portfolio). To impose that constraint we would use arguments:

```
dist.center=targwt, dist.style="weight", dist.bounds=.5
```

The example above computes distance as the difference in weights. The distance can alternatively be computed as the difference in monetary values. These two are equivalent if the gross value is constrained to a narrow range (although the weight formulation takes slightly more time to compute). However, if the gross value is allowed a wide range, then there is a difference: the value distance will favor a gross value close to the gross value of the target portfolio while the weight distance will be indifferent to the gross value.

For a value distance, you can give either the values of the assets in the target portfolio or the shares in the target portfolio. Using value might look like:

```
dist.center=target.values, dist.style="value",
dist.bounds=25000
```

while using shares might look like:

```
dist.center=target.shares, dist.style="shares",
dist.bounds=25000
```

The `dist.trade` argument takes a logical vector and allows you to constrain the distance of the trade rather than the portfolio.

The `dist.utility` argument also takes a logical vector—`TRUE` values of this imply that you want to use it as the objective in optimization. For more on this, see Section 5.4.

Alternative prices

By default, distance is defined by the `prices` vector. You don't have to define it that way. You can include a value for `dist.prices` that gives different prices to be used in the computations of distances. The `dist.style` argument takes a value with "custom" before the style when you want to use the `dist.prices` values. For example, you might have:

```
dist.center=target.shares, dist.style="customshares",
dist.prices=some.price.vector, dist.bounds=35000
```

Multiple distances

It is possible to have more than one distance constraint. In this case the `dist.center` argument needs to be a list. For example, you might have arguments that look like:

```
dist.center=list(target1.wt, target2.share, target3.wt),
dist.style=c("weight", "shares", "weight"),
dist.bounds=c(0.2, 2e5, 0.3)
```

Here we have three distances declared—the first and third in terms of weight and the second in terms of shares.

If you want lower bounds as well as upper bounds on the distances, then you need to give `dist.bounds` a two-column matrix with each row corresponding to a distance. For example:

```
dist.bounds=cbind(c(0.1, 1e5, 0.2), c(0.2, 2e5, 0.3))
```

Each of the arguments `dist.style`, `dist.trade` and `dist.utility` can either have length equal to the number of distances, or length one if the argument is the same for all arguments. For example, in the example above `dist.trade` and `dist.utility` both have their length one default value.

The `dist.prices` argument can be a list of price vectors to match the `dist.center` argument.

3.14 Sums of Largest Weights

This section discusses the argument:

- `sum.weight`

Consider the constraint that the largest n weights should sum to no more than some limit—this is handled by the `sum.weight` argument.

For example if you want the 5 largest weights to sum to no more than 40%, and the 10 largest weights to sum to no more than 70%, the argument would look like:

```
sum.weight = c("5"=.4, "10"=.7))
```

The values in the vector given to `sum.weight` are the limits, the names of the vector are the numbers of weights in the sums.

S code note

The names within the `c` function must be inside quotes as they are not legal S object names.

Another way of getting the same thing would be:

```
> sumw.arg <- c(.4, .7)
> names(sumw.arg) <- c(5, 10)
```

Then `sumw.arg` would be given as the `sum.weight` argument.

The argument:

```
sum.weight = c("1" = .05)
```

creates a constraint that says that the largest weight of any one asset is 5%. This is conceptually exactly the same as:

```
max.weight = .05
```

However, there are some operational differences. The key difference is that using `sum.weight` ensures that either the constraint is satisfied or there is a penalty imposed for breaking the penalty. But it will be somewhat slower when `sum.weight` is used. See page 32 for the possibility of `max.weight` failing to enforce the constraint.

So far only upper bounds have been discussed. It is possible to impose lower bounds as well as upper bounds. This is done by giving a two-column matrix where the first column is the lower bound and the second column is the upper bound. The row names give the number of the largest weights.

A couple examples are:

```
> rbind('5'=c(.2, .3))
  [,1] [,2]
5  0.2  0.3
> rbind('5'=c(.2, .3), '10'=c(.3, .4))
   [,1] [,2]
5   0.2  0.3
10  0.3  0.4
```

Using the first matrix would restrict the sum of the largest 5 weights to be between 20% and 30%. The second matrix would add the constraint that the sum of the largest 10 weight have to sum to between 30% and 40%.

3.15 Cost Constraints

This section discusses the argument:

- `limit.cost`

This is unlikely to be of use with optimization.

`limit.cost` puts lower and upper constraints on the cost. This can be useful for mimicking actual trading with random portfolios. If this is given, then it must be a length two vector giving the allowable range of costs:

```
limit.cost = c(2.3, 2.35)
```

or possibly something like:

```
limit.cost = opt.obj$results["cost"] * c(.99, 1.01)
```

3.16 Number of Positions to Close

This section discusses the argument:

- `close.number`

The minimum and maximum number of positions to close in the existing portfolio can be specified with the `close.number` argument. If a single number is given, then exactly that number are to be closed.

So:

```
close.number = 3
```

is really shorthand for:

```
close.number = c(3, 3)
```

Note that this is different than other arguments where a single number does not imply a lower bound.

To specify that at least 2 positions should be closed, but no more than 3, then say:

```
close.number = c(2, 3)
```

This sort of constraint is unlikely to be useful in optimization, but can be used to generate random portfolios that match what an actual trade has done.

3.17 Quadratic Constraints

Quadratic constraints on the portfolio are not officially supported, but some can be imposed. Quadratic constraints on the trade are not possible.

The acceptance of more than one variance means that `trade.optimizer` and `random.portfolio` can be tricked into allowing quadratic constraints on the portfolio. However, general constraints are not possible – only ones where the matrix is symmetric (because the algorithm speeds computations by assuming the variance is symmetric).

Here are the steps for creating quadratic constraints:

- Add the constraint matrices to the variance.
- Specify the constraint bounds.
- Make sure the constraints are not benchmark-relative.
- If doing optimization, tell the optimizer not to include the constraints in the utility.

Add Constraints to the Variance

Suppose you have a single variance matrix (as per usual) called `varian` and you have two quadratic constraint matrices called `qmat1` and `qmat2`. You need to create a three dimensional array with three slices. This is an easy operation in the S language, but care needs to be taken that the assets match.

```
> assetnam <- dimnames(varian)[[1]]
> varian3 <- array(c(varian, qmat1[assetnam, assetnam],
+    qmat2[assetnam, assetnam]), c(dim(varian), 3),
+    c(dimnames(varian), list(NULL)))
```

S code note

The `c` function features prominently in this command.

When `c` is used on matrices, as in its first occurrence in the command, the dimensions are stripped and the result is a plain vector of all of the elements of the three matrices.

The `c` function can be used with lists as well as with atomic vectors. For the dimnames of the resulting three-dimensional array, we need to have a length three list where the final component is ignored (and hence can be NULL). In this case we need all of the arguments to `c` to be lists, so we need to give it `list(NULL)`. Just giving an argument of `NULL` would not have had the effect that we wanted.

Impose Constraint Bounds

The next step is to impose the constraint bounds. This will be done with the `var.constraint` argument as discussed in Section 13.3. The constraints in this case will be on the second and third variances, that is, on indices 1 and 2. Also note that the bounds are in terms of the weights—there is not a choice to put the bounds in terms of monetary value.

If you are generating random portfolios (without a utility constraint) and you are not using a benchmark, then you can proceed to the actual computation. The intervening steps only apply if you are using a benchmark or a utility.

Dummy Run

The next two steps are aided by performing a dummy run of optimization—something like:

```
> qc.temp <- trade.optimizer(prices, varian3,
+    iterations=0, ...)
```

It is best to include the variance constraints in this run—without them the step regarding benchmarks can go wrong. When `iterations.max` is set to zero, then a minimal amount of optimization is performed. You will get a warning that this is not the same as no optimization. We don't care about optimization in this instance, we only care about the setup for optimization.

Check for Benchmark

If you are not using a benchmark, you don't need to be concerned with this step.

If you are using a benchmark, then the default behavior makes the variances relative to the benchmark. This is seen in the `vtable` component of the output of our dummy run.

```
> qc.temp$vtable
             [,1] [,2]  [,3]
variance        0    1     2
```

```
benchmark     500  500   500
utility.only    1    0     0
attr(,"benchmarks")
[1] "spx" "spx" "spx"
```

In our example we want the last two columns to not have a benchmark. Hence we would do an operation like:

```
> qcvtab <- qc.temp$vtable
> attr(qcvtab, "benchmarks") <- c("spx", "", "")
```

If you are generating random portfolios and you didn't include the variance constraints in the dummy run, then you will also want to change the `utility.only` row to be zero in the columns that pertain to your constraints. The argument:

```
vtable = qcvtab
```

should be given in the actual computation.

You can learn more about the `vtable` argument on page 126.

Constraints out of Utility

This step does not pertain to random portfolios (unless a utility constraint is used).

This step tells the optimizer that the final two "variances" are not to be used in the utility. We do this by modifying the utility table from the dummy run. (Utility tables are discussed on page 131.)

```
> qc.utiltab <- qc.temp$utable
> qc.utiltab
                  [,1] [,2] [,3]
alpha.spot           0    0    0
variance.spot        0    1    2
destination          0    1    2
opt.objective        1    1    1
risk.aversion        1    1    1
wt.in.destination    1    1    1
> qc.utiltab['destination', ] <- c(0, -1, -1)
> qc.utiltab
                  [,1] [,2] [,3]
alpha.spot           0    0    0
variance.spot        0    1    2
destination          0   -1   -1
opt.objective        1    1    1
risk.aversion        1    1    1
wt.in.destination    1    1    1
```

Actual Computation

Now we are ready for the actual computation. If we are generating random portfolios, we'll have a command like:

```
> qc.rp <- random.portfolio(100, prices, varian3,
+    vtable = qcvtab, var.con = ...)
```

Alternatively, if we are optimizing, we'd do something like:

```
> qc.opt <- trade.optimizer(prices, varian3,
+    vtable = qcvtab, utable = qc.utiltab, var.con = ...)
```

3.18 Constraint Penalties and Soft Constraints

This section applies only to optimization—it can not apply to generating random portfolios.

Virtually all of the constraints are enforced via penalties. That is, instead of the optimization algorithm minimizing the negative utility, it minimizes the negative utility plus penalties for whatever constraints are violated. Solutions that obey all of the constraints experience no penalties.

The algorithm actively attempts to satisfy some of the constraints—in particular the constraints on the monetary value of portfolios are closely managed, as are threshold constraints, and the constraint on the maximum number of assets traded is always obeyed.

Tools are available in `trade.optimizer` to allow you to adjust the penalties so that most of the constraints (those not actively managed) can become soft constraints. This is done by making the penalties for those constraints small enough so that there will be a trade-off between the penalty for the constraint and the utility.

Consider an example:

```
> soft.op1$violated
[1] "variance"    "linear"      "gross value"
attr(,"linear violations")
[1] "Country" "Sector"
```

There are violations of two linear constraints, the variance constraint, and the gross value.

The default value of the `penalty.constraint` argument is 1000. You can change individual elements of the penalty by giving the `penalty.constraint` argument a named vector where the names are the items for which you want to change the penalty. You need to give the names exactly—they can not be abbreviated. The easiest approach is to change the `penalty.constraint` component of the output from the original optimization.

```
> softpen2 <- soft.op1$penalty.constraint
> softpen2[1:2] <- .1
> soft.op2 <- update(soft.op1, penalty.constraint=softpen2)
```

Chapter 4

Valuation of Portfolios

In addition to generating random portfolios and optimizing trades, the third major computational task in Portfolio Probe is the valuation of portfolios.

The `valuation` function is generic and has three methods:

- the default method takes a named numeric vector representing the portfolio, and it requires prices to be given.
- There is a method for results from `trade.optimizer`. This will use the prices in the original call if prices are not given.
- There is a method for random portfolios; this requires prices to be given.

4.1 Single Portfolio

A single portfolio is valued via the default `valuation` method. The method for optimization objects mostly just does the same thing as the default. This section covers both these cases.

If `opti` is the result of `trade.optimizer`, you get valuation of the portfolio with the first command below, and of the trade with the second:

```
> valuation(opti)
> valuation(opti, trade=TRUE)
```

The default behavior of `valuation` when given an optimization object is to give information on the portfolio. The trade can be examined instead by setting the `trade` argument to `TRUE`.

When `valuation` is given a vector (and hence the default method is used), prices must be given:

```
> valuation(opti$trade, price=new.prices)
```

Prices are optional when optimization objects are given—the default is the prices used during the optimization.

Summary Statistics

By default several pieces of information are given in the result of a `valuation` call. For instance:

```
> valuation(opti)
$individual
       D       E 
   28.95 9970.40 
$total
  gross     net    long   short 
9999.35 9999.35 9999.35    0.00 
$weight
           D           E 
 0.002895188 0.997104812 
$timestamp
[1] "Mon Feb 20 18:35:36 2012"
$call
valuation.portfolBurSt(x = opti)
```

One way to get the gross value of the portfolio is:

```
> valuation(opti)$total["gross"]
```

Another way is:

```
> valuation(opti, collapse=TRUE, type="gross")
```

Weights

You can get the asset weights by extracting the `weight` component:

```
> valuation(opti)$weight
          D           E 
0.002895188 0.997104812 
```

This effectively drops zero weights from the universe. Often we want the weight vector to correspond to the full universe. The `all.assets` argument gets us that:

```
> round(valuation(opti, the.pri, all.assets=TRUE)$weight, 4)
     A      B      C      D      E      F 
0.0000 0.0000 0.0000 0.0029 0.9971 0.0000 
```

The assets in the resulting weight vector will be in the same order as the assets in the prices vector. Notice that it is useless to specify `all.assets=TRUE` and not give `prices` since the prices in an optimization object are only a selection of the universe.

Collapsing Values

So far we have presumed that `prices` is just a vector of the prices at one time. But `prices` in `valuation` can be:

- a vector — prices of the assets at a specific time
- a matrix — rows corresponding to different times, columns corresponding to different assets
- a 3-dimensional array — rows corresponding to different times, columns corresponding to different assets, slices corresponding to different scenarios

If `prices` is a matrix or an array, then `collapse` can not be `FALSE`. (See Section 4.4 for values of `collapse` other than `TRUE` and `FALSE`.)

The `type` argument comes into play when `collapse` is not `FALSE`. The possible values of `type` are:

- `"gross"` — the gross value
- `"net"` — the net value
- `"long"` — the value of the long positions
- `"short"` — the value of the short positions (a positive number)
- `"nav"` — the net asset value (which includes `cash`)
- `"cash"` — the value(s) computed for cash

The net asset value is the net value plus the supporting cash. If the `cash` argument is not given, then cash is computed to be the gross value minus the net value (at the first time point). So the cash of a long-only portfolio is computed to be zero, and the cash of a dollar neutral portfolio defaults to the gross value.

It is possible to have a command on the order of:

```
> valuation(opti, collapse=TRUE, type="gross", weight=TRUE)
```

In this case it is a long way around to get to 1. However, if `type` were one of the other possibilities and the portfolio were long-short, then it could be of interest.

4.2 Random Portfolios

Valuation of random portfolios is very similar to that for single portfolios, but there are a few differences. Specifically the random portfolio method gives valuations for multiple portfolios, not just one.

When collapse is FALSE

When `collapse` is `FALSE`, then the result is a list that contains the valuation or weights of the assets in the portfolios. Each component corresponds to one of the random portfolios.

The single portfolio methods also return a list in this case, but that is a list with multiple components for the one portfolio.

The `prices` argument is allowed to be a matrix in this case for random portfolios (but it is not currently allowed to be three-dimensional). For the single portfolio methods `prices` can only be a vector when `collapse` is `FALSE`.

When collapse is not FALSE

When `collapse` is `TRUE`, then the result is an object much like the `prices` that is input. The difference is that random portfolios replace the assets. So if `prices` is a matrix of times by assets, then the result will be a matrix of times by random portfolios.

See Section 4.4 for the remaining possibility for `collapse`.

4.3 Compute Returns

In addition to returning monetary values or weights, it is also possible to return returns. This requires that there be prices for more than one time period—that is, that `prices` be a matrix or three-dimensional array.

When the `returns` argument is `NULL`, then values or weights are returned (depending on the `weight` argument). The other possible values of `returns` are:

- `"simple"`
- `"log"`
- `"arithmetic"`
- `"geometric"`

This is not four different types of returns, this is two types of returns that each have two names. The result will be an object similar to what would be returned if valuations were asked for, but it will have one less time point.

If returns are requested, then `collapse=FALSE` and `weight=TRUE` are overridden. Also `type` is set to `"nav"`.

4.4 Collapse into Categories

There is a half-way case of collapsing, which is to collapse into categories of assets.

All of the possible values of `collapse` are:

- `FALSE`
- `TRUE`

- a named character vector
- a named factor
- a list of named character vectors and/or named factors
- a data frame of character and/or factor columns where the row names of the data frame are the asset identifiers

The names on the character vectors or factors are the asset identifiers. In the example where asset identifiers are single uppercase letters, such a character vector might be:

```
> sector.char
       A        B        C        D        E         F
"Retail" "Retail" "Energy" "Energy" "Mining"  "Media"
```

There is no problem having assets in the category objects that are not used.

A list of categories is used in order to get combinations of the categories. For example if we had:

```
collapse = list(sector.char, country.factor)
```

Then we would end up with categories like `"Retail.UK"` and `"Media.France"`.

If `prices` is a vector, then the result for single portfolios will be a vector corresponding to categories, and the result for random portfolios will be a matrix that is categories by random portfolios.

If `prices` is a matrix, then the result for single portfolios will be a matrix that is time by categories, and for random portfolios it will be a three-dimensional array that is time by categories by random portfolios. Note that it is valid to ask for returns in this case.

Computations with categories and three-dimensional `prices` are not currently implemented.

4.5 Summary

Table 4.1 summarizes the use of `valuation`. The codes for `prices` are: v for vector, m for matrix and 3 for three-dimensional array. The codes used for the results are explained in Table 4.2. A star (*) means any allowable input for that particular argument.

Table 4.1: Summary of possibilities with `valuation`.

portfolio	**prices**	**collapse**	**weight**	**returns**	**result**
single	v	F	*	NULL	list of info
single	*	T	F	NULL	pval(0D/1D/2D)
single	*	T	T	NULL	pwt(0D/1D/2D)
single	m, 3	F, T	*	string	pret(1D/2D)
single	v, m	categ	F	NULL	cval(1D/2D)
single	v, m	categ	T	NULL	cwt(1D/2D)
single	m	categ	*	string	cret(2D)
random	v, m	F	F	NULL	list of aval(1D/2D)
random	v, m	F	T	NULL	list of awt(1D/2D)
random	v, m, 3	T	F	NULL	pval(1D/2D/3D)
random	v, m, 3	T	T	NULL	pwt(1D/2D/3D)
random	m, 3	F, T	*	string	pret(2D/3D)
random	v, m	categ	F	NULL	cval(2D/3D)
random	v, m	categ	T	NULL	cwt(2D/3D)
random	m	categ	*	string	cret(3D)
random	3	F	*	NULL	Error
single	m, 3	F	*	NULL	Error
*	v	*	*	string	Error
*	3	categ	*	*	Error

Table 4.2: Explanation of the "results" column in Table 4.1.

code	**explanation**
aval	asset valuation
awt	asset weight
pval	portfolio valuation
pwt	portfolio weight
pret	portfolio return
cval	category valuation
cwt	category weight
cret	category return

Chapter 5

Optimizing Long-Only Portfolios

This chapter discusses commands for optimizing long-only portfolios. It applies to both active and passive portfolios.

The examples in this chapter are just sketches of what you are likely to want to do in practice. Chapter 3 discusses all the constraints that may be imposed. Some of the examples limit the number of assets that are held in the portfolio or traded. These limits are convenient, but threshold constraints (page 37) may be better to use. Chapter 8 covers trading costs—it is usually a good idea to constrain trading in some way, whether with trading costs or a turnover constraint.

5.1 Required Inputs

The `trade.optimizer` function needs a price vector. The `prices` argument needs to be a vector of prices with names that are the asset names. The assets in `prices` control the assets that are used in the problem. Other inputs—such as the variance and linear constraints—may contain additional assets (which will be ignored).

Other than `prices` there is no single argument that is required. However, there are two concepts that are required: the amount of money in the portfolio has to be constrained somehow, and there has to be something that provides a utility.

Monetary Value

The turnover or the amount of money in the portfolio needs to be constrained. Details can be found on page 27

Utility

At least one of the arguments:

- `variance`

- `expected.return`
- `dist.center`

must be given.

Utility-free optimization—minimizing the distance to a target portfolio—can be done using `dist.center` and related arguments. See Section 5.4 to do this.

Assuming distance is out of the picture: If `expected.return` is not given, then the utility is the minimum variance. If a variance matrix is not given, then the utility is the maximum expected return. If both are given, then the default utility is to maximize the information ratio. It can be safest to explicitly state the utility that you want.

Optimizing a long-only portfolio without a variance or a distance is very unlikely to be a good idea.

5.2 Examples for Passive Portfolios

This section contains some examples that build in complexity. Expected returns do not enter into optimization for passive portfolios, and they need not be given.

Minimize the Variance of the Portfolio

This is the simplest optimization to perform:

```
> op.mv1 <- trade.optimizer(prices, varian,
+    utility = "minimum variance",
+    long.only=TRUE, gross.value=1.4e6)
```

The amount put into the portfolio is 1.4 million currency units. The simplicity of this command means that it may not do what you want since it is dependent on default values for most arguments.

Minimize Tracking Error

We can minimize the variance relative to a benchmark—that is, minimize tracking error. To be concrete, let's assume that the benchmark is the S&P 500 with the asset name of `"spx"`.

The benchmark need not be in the price vector unless trading the benchmark is allowed (via the `bench.trade` argument).

A simple version of minimizing the tracking error is:

```
> op.te1 <- trade.optimizer(prices, varian,
+    utility = "minimum variance",
+    long.only=TRUE, gross.value=1.4e6, benchmark="spx",
+    bench.weights=list(spx=spx.wts))
```

The `bench.weights` argument is used to add the benchmark to the variance (see page 92 for more details). It is not necessary if the benchmark is already incorporated into the variance.

This is almost surely too simple.

For example, you may want to limit the number of assets in the portfolio:

```
> op.te2 <- trade.optimizer(prices, varian,
+    utility = "minimum variance",
+    long.only=TRUE, gross.value=1.4e6, benchmark="spx",
+    port.size=55, max.weight=0.05,
+    bench.weights=list(spx=spx.wts))
```

The `op.te2` object represents a portfolio containing up to 55 assets that (approximately) minimizes the tracking error among all portfolios of size 55 in which assets have a maximum weight of 5%.

Constraining the maximum weight is really a proxy for constraining risk. We can do this directly with the `risk.fraction` argument (see page 33 for details).

```
> op.te3 <- trade.optimizer(prices, varian,
+    utility = "minimum variance",
+    long.only=TRUE, gross.value=1.4e6, benchmark="spx",
+    port.size=55, risk.fraction=0.05,
+    bench.weights=list(spx=spx.wts))
```

In this case the `bench.weights` argument is necessary even if the benchmark is in the variance – the risk fraction computation requires the benchmark weights.

If you are rebalancing an existing portfolio rather than creating a new one, then the command might be:

```
> op.te4 <- trade.optimizer(prices, varian,
+    utility = "minimum variance",
+    long.only=TRUE, gross.val=1.4e6, benchmark="spx",
+    risk.fraction=0.05, existing=cur.port, ntrade=20,
+    bench.weights=list(spx=spx.wts))
```

The command creating `op.te4` is allowing up to 20 assets to be traded. It does not control the turnover, nor the number of assets in the resulting portfolio. Thus `op.te4` can potentially have up to 75 assets in it if the current portfolio contains 55 assets. Suppose that we want to have a portfolio with 45 to 55 assets in it, and we want to hold the turnover to 100,000 (currency units, e.g., dollars). The following command will provide that:

```
> op.te5 <- trade.optimizer(prices, varian,
+    utility = "minimum variance",
+    long.only=TRUE, gross.value=1.4e6, benchmark="spx",
+    risk.fraction=0.05, existing=cur.port, turnover=1e5,
+    port.size=c(45,55), bench.weights=list(spx=spx.wts))
```

This command is a realistic possibility.

The examples above could have been done without specifying the `utility` argument—since there are no expected returns, the utility would default to minimum variance (but you would get a warning stating that utility was being modified).

See page 78 for examples of dealing with cash flow. Page 132 has an example of minimizing against dual benchmarks—for example, when you have a prediction of the benchmark after an upcoming rebalancing.

5.3 Examples for Active Portfolios

Expected returns of the assets are required for optimization of active portfolios. (This is not absolutely true—see page 77 for an example.) Additionally, there needs to be a decision on how to balance expected return with risk.

Maximize the Information Ratio

The default optimization that involves both expected returns and variance is to maximize the information ratio. The information ratio of a portfolio is the expected return of the portfolio divided by the standard deviation of the portfolio. (The standard deviation is the square root of the variance.) This is a natural quantity (in mean-variance land) to maximize—we're seeking the best risk-adjusted return.

A command to maximize the information ratio is:

```
> op.ir1 <- trade.optimizer(prices, varian,
+    long.only=TRUE, expected.return=alphas,
+    gross.value=1.4e6)
```

The `alphas` object is a vector of expected returns of the assets. When both variance and expected returns are given, the default objective is to maximize the information ratio. In this example 1.4 million currency units are put into the portfolio.

The information ratio is traditionally put into annualized terms. If the variance and expected returns in your command are not annualized, then you can annualize the information ratio by multiplying the utility by the square root of the number that you would use to annualize the variance and expected returns. For example, for a daily frequency the annualized information ratio would be:

```
> -op.ir1$utility.value * sqrt(252)
```

The negative sign is because the optimizer always minimizes. When quantities are being maximized (information ratio or utility), then it really minimizes the negative of the quantity.

Note that this is the information ratio you would get if all of your inputs were exact. The actual information ratio that you achieve is likely to be worse.

We now turn to the case of maximizing the information ratio when there is a benchmark involved.

The Information Ratio with a Tracking Error Constraint

It is common to perform a mean-variance optimization relative to the benchmark. The risk aversion is adjusted so that the tracking error is near its desired level. It makes more financial sense to maximize the information ratio in absolute terms (as in the command that created `op.ir1`) while having a constraint that the tracking error is no larger than a given value.

The mechanism in `trade.optimizer` that allows a constraint relative to a benchmark is the `bench.constraint` argument. Here is a command to constrain the tracking error to be no more than 4% while maximizing the information ratio:

```
> op.ir2 <- trade.optimizer(prices, varian,
+    long.only=TRUE, expected.return=alphas,
+    gross.value=1.4e6, bench.constraint = c(spx=0.04^2/252))
```

The command above presumes that the benchmark is already included in the variance matrix. If it isn't, then the `bench.weights` argument could be used. It would be something like:

```
bench.weights = list(spx=spx.weight.vector)
```

The value of the `bench.constraint` argument needs to be named (so that it knows what benchmark you want) and the value is in the scale of the variance. Using the `c` function allows a name to be put on the value.

The 0.04 is squared to transform the tracking error into a variance. This is then divided by 252 in order to go from the annual value to daily, which is the frequency of the variance we are assuming in this example. If the variance were annualized and created from returns in percent, then the `bench.constraint` argument would have been:

```
c(spx=4^2)
```

note

The `benchmark` argument is not given in the previous command. If it had been, then the information ratio that was optimized would be relative to the benchmark.

A more realistic command might be:

```
> op.ir3 <- trade.optimizer(prices, varian,
+    long.only=TRUE, expected.return=alphas,
+    gross.value=1.4e6, turnover=1e5, ntrade=15,
+    bench.constraint = c(spx=0.04^2/252),
+    existing=cur.port,  port.size=60)
```

This command includes the existing portfolio, has a limit on the turnover, and specifies that the final portfolio should contain no more than 60 assets.

You can easily add constraints to other benchmarks as well—just make the `bench.constraint` argument longer:

```
> op.ir4 <- trade.optimizer(prices, varian,
+    long.only=TRUE, expected.return=alphas,
+    gross.value=1.4e6, turnover=1e5, ntrade=15,
+    bench.constraint = c(spx=0.04^2/252, newspx=0.05^2/252),
+    existing=cur.port, port.size=60)
```

In this example we are constraining the tracking error to be no more than 4% against the current benchmark, and no more than 5% against our prediction of the benchmark after it is rebalanced.

Minimizing relative to two or more benchmarks is, however, somewhat more complicated—that is discussed on page 132. Also a little more complicated is to get a benchmark-relative utility but a constraint on the portfolio variance—page 134 shows how to do that.

Maximize Benchmark-relative Information Ratio

If you want the the utility—in this case the information ratio—to be relative to a benchmark, then you merely need to give the `benchmark` argument:

```
> op.ir5 <- trade.optimizer(prices, varian,
+    long.only=TRUE, expected.return=alphas,
+    gross.value=1.4e6, benchmark = "spx")
```

You get substantially different behavior depending on whether you use `benchmark` or `bench.constraint`:

- `bench.constraint` does not affect which utility is used.
- `benchmark` changes the utility.

You can use both—in which case `benchmark` changes the utility as usual, but there is also a constraint on the tracking error.

Mean-Variance Optimization

The default objective is to maximize an information ratio (whenever both variance and expected returns are given). If you want to perform a mean-variance optimization instead, then you need to specify the `utility` argument. The generic mean-variance problem is to maximize the expected return of the portfolio minus the risk aversion times the variance of the portfolio. Page 123 has more on the risk aversion parameter.

A command to do a simple mean-variance optimization is:

```
> op.mv1 <- trade.optimizer(prices, varian,
+    long.only=TRUE, expected.return=alphas,
+    gross.value=1.4e6, utility='mean-var', risk.aversion=.7)
```

This command does not include a benchmark. As noted above, if the `benchmark` argument is given, then the mean and variance will be relative to the benchmark. If you do have a benchmark, an alternative approach is to do the optimization in the absolute sense and add a constraint on the tracking error. The tracking error constraint is placed on the optimization with a command like:

```
> op.mv2 <- trade.optimizer(prices, varian,
+    long.only=TRUE, expected.return=alphas,
+    gross.value=1.4e6, utility='mean-var', risk.aversion=.7,
+    bench.constraint = c(spx=0.04^2/252))
```

For an explanation of the value given for `bench.constraint`, see page 56.

In mean-variance optimization, the risk aversion parameter is quite important.

Mean-Volatility Optimization

Mean-volatility optimization is just like mean-variance optimization except that the value of the `utility` argument is:

```
utility = "mean-volatility"
```

and the `risk.aversion` argument will (usually) have a substantially different value.

Buy-Hold-Sell List

It is not absolutely necessary to create expected returns for an actively managed portfolio. Instead you can minimize the variance or tracking error using a list of assets that can be sold and a list of assets that can be bought.

Suppose that `buy.vec` and `sell.vec` are vectors giving the assets that are allowed to be bought and sold. The first thing to do is to set up the constraints that make sure nothing on the buy list can be sold, and nothing on the sell list can be bought.

```
> lower.tr <- rep(0, length(buy.vec))
> names(lower.tr) <- buy.vec
> upper.tr <- rep(0, length(sell.vec))
> names(upper.tr) <- sell.vec
```

S code note

These commands create two new vectors containing all zeros and with names that are the buy assets for one of them, and names that are the sell assets for the other. The `rep` function replicates its first argument.

Then these are used in the call to the optimizer. In this case we assume that we want to minimize tracking error relative to `spx`.

```
> op.bhs <- trade.optimizer(prices, varian,
+     long.only=TRUE, gross.value=1.4e6, benchmark="spx",
+     existing=cur.port, lower.trade=lower.tr,
+     upper.trade=upper.tr,
+     universe.trade=c(buy.vec, sell.vec))
```

The `universe.trade` argument is restricting the assets that are traded to be only in the buy or sell list. Assets not named are constrained to not trade.

You may wish to use the `turnover` argument to constrain turnover, and other arguments.

5.4 Utility-free Optimization

The idea of utility-free optimization is that you have an ideal target portfolio and you would like to get as close to that target as possible while still obeying all of your portfolio constraints. Expected returns are not used for this and you don't have to have a variance matrix.

One of the simplest practical possibilities is a call like:

```
> op.d1 <- trade.optimizer(prices, dist.center=target.wts,
+    utility="distance", existing=current.portfolio,
+    turnover=30000, gross.value=1e6, long.only=TRUE)
```

The key parts of this call are the `dist.center` argument that takes a representation of the target portfolio, the `utility` argument, and the `turnover` argument. The call is taking advantage of the default value of `dist.style` which is `"weight"`. The `target.wts` object should be a named numeric vector that sums to 1, where the names are asset ids.

Some sort of constraint on turnover is almost surely desirable. That could be accomplished using costs, but the `turnover` argument is a much easier option.

Another constraint that is quite likely in this setting is a tracking error constraint. For this we do need a variance and the `bench.constraint` argument must be specified.

```
> op.d2 <- trade.optimizer(prices, dist.center=target.wts,
+    utility="distance", existing=current.portfolio,
+    turnover=30000, gross.value=1e6, long.only=TRUE,
+    variance=varian, bench.constraint=c(spx=0.04^2/252))
```

In this case we are specifying a maximum tracking error of 4% assuming the variance was produced with daily data—see Section 3.12 for more details.

The default style of distance (as controlled by the `dist.style` argument) is `"weight"`. In this style distance is considered to be the sum of absolute differences in the weights of the new portfolio and the target portfolio. This is often how we conceive of distance, but it has the disadvantage that the answer need not be unique. An option is to add the argument:

```
dist.style="sumsqwi"
```

This changes the distance to be the weighted sum of squared differences in weights, where the weights (for the sum of squares) are the inverse of the weights for the target portfolio. (Assets not in the target portfolio are ignored in this formulation of distance.) This form of distance may or may not be more pleasing.

Section 3.13 has more information on the arguments that control distance.

5.5 Managing Cash Flow

When you either are required to raise cash or have cash to put into the portfolio, you can use optimization to find a good trade. If there is significant cash flow, then this process has the benefit of keeping the portfolio closer to optimal than it otherwise would be—thus reducing the need for large, periodic rebalances. The examples given below assume an active portfolio—merely eliminate the `expected.return` argument if yours is passive.

Injecting Money into a Portfolio

If you want to put more money into an existing portfolio, then the main thing to say is that you don't want any selling. You probably also want to limit the number of assets to trade as well. One possible command is:

```
> op.in1 <- trade.optimizer(prices, varian,
+    long.only=TRUE, expected.return=alphas, turnover=newcash,
+    bench.constraint = c(spx=0.04^2/252),
+    existing=cur.port, ntrade=5, lower.trade=0)
```

The key features in this command are setting the `turnover` argument to the amount of money to be injected, and setting the `lower.trade` argument to zero.

You may also want to use the `port.size` argument to restrict the number of assets in the new portfolio. Alternatively you could set the `universe.trade` argument to be the names in the current portfolio—meaning that no additional assets will be added to the portfolio.

Extracting Money out of a Portfolio

To raise cash from the portfolio, you want the optimizer to do no buying. This is achieved with the following command:

```
> op.out1 <- trade.optimizer(prices, varian,
+    long.only=TRUE, expected.return=alphas,
+    turnover=cash.required, ntrade=5,
+    bench.constraint = c(spx=0.04^2/252),
+    existing=cur.port, upper.trade=0)
```

The pertinent parts of this command are setting the `turnover` argument to the amount of cash that is required, and setting `upper.trade` to zero.

The formulations given for both injecting and extracting money are implicitly assuming that the optimizer will want to trade at least the amount requested. That doesn't have to be true. A safer approach would be to give `turnover` an interval, perhaps something like:

```
turnover = cash.required * c(1, 1.01)
```

5.6 Asset Allocation

Asset allocation as we discuss it here means that we are interested in weights rather than prices and asset units (like shares or lots). We restrict asset allocation to long-only.

`trade.optimizer` can handle such problems.

The key step is to create a price vector that is all ones.

```
> aa.prices <- rep(1, length(asset.names))
> names(aa.prices) <- asset.names
```

You then select a gross value that is large enough to satisfy the accuracy that you desire. For example if getting the weight to the nearest basis point is good enough for you, then set the gross value to 10,000. The gross value should be given as an interval that has just one integer in it.

The command:

```
> aa.alt1 <- trade.optimizer(aa.prices, avar,
+    expected.return=aret, utility='mean-var',
+    risk.aversion=.02, long.only=TRUE,
+    gross.value=10000+c(-.5, .5))
```

performs a mean-variance optimization.

You can get the weights from the portfolio optimizer object with:

```
> valuation(aa.alt1)$weight
```

A further example of this technique is the multiple scenario analysis on page 138.

5.7 Going Farther

Tasks that you might want to undertake:

- To add trading costs to your optimization command, see Chapter 8.
- To add constraints, see Chapter 3 on page 27.
- To review common mistakes, see Section 9.1.
- To improve your solution or examine the quality of solutions you are getting, go to Chapter 11 on page 115.
- To export your solution to a file, see Section 7.3.
- To perform optimizations with multiple variances, expected returns and/or benchmarks, go to Chapter 13.

Chapter 6

Optimizing Long-Short Portfolios

This chapter discusses commands to optimize long-short portfolios.

The examples in this chapter are just sketches of what you are likely to want to do in practice. Chapter 3 discusses all the constraints that may be imposed. Some of the examples limit the number of assets that are held in the portfolio or traded. These limits are convenient, but threshold constraints (page 37) may be better to use. Chapter 8 covers trading costs—it is usually a good idea to constrain trading in some way, whether with trading costs or a turnover constraint.

6.1 Required Inputs

The `trade.optimizer` function needs a price vector. The `prices` argument needs to be a vector of prices with names that are the asset names. The assets in `prices` control the assets that are used in the problem. Other inputs—such as the variance and linear constraints—may contain additional assets (which will be ignored).

Other than `prices` there is no single argument that is required. However, there are two concepts that are required: the amount of money in the portfolio has to be constrained somehow, and there has to be something that provides a utility.

Monetary Value

The turnover or the amount of money (both long and short) in the portfolio needs to be constrained. Details can be found on page 27

Utility

At least one of the arguments:

- `variance`

- `expected.return`
- `dist.center`

must be given.

The `dist.center` argument and its mates are used to do utility-free optimization— that is, to minimize the distance to an ideal target portfolio. This is likely to be more common for long-only portfolios, but there is no reason not to do it for long-short portfolios as well. If you do want to do it, see Section 5.4 (which assumes long-only).

Assuming distance is out of the picture: If `expected.return` is not given, then the utility is the minimum variance. If a variance is not given, then the utility is the maximum expected return. If both are given, then the default utility is to maximize the information ratio.

6.2 Examples

This section presents some examples. Feel free to skip over those that don't apply to you. These examples do not include arguments for the cost of trading (see Chapter 8). Nor does it cover several of the constraints (see Chapter 3).

Maximize the Information Ratio

If you have a set of expected returns and a variance, then maximizing the information ratio is a natural thing to do— and it is the default. This can be done simply, with a command similar to:

```
> opt.ir1 <- trade.optimizer(prices, varian,
+    expected.return=alphas, gross.value=1.4e6,
+    net.value=c(-1e4, 2e4))
```

This puts 1.4 million currency units into the gross value of the portfolio, and the net value is between negative 10,000 and positive 20,000.

There are a few additional arguments that will be useful in practice. If you are building a new portfolio, then a more likely command is something like:

```
> opt.ir2 <- trade.optimizer(prices, varian,
+    expected.return=alphas, gross.value=1.4e6,
+    net.value=c(-1e4, 2e4), port.size=55, max.weight=.05)
```

This new command adds `port.size` to give a portfolio containing up to 55 assets. It also ensures that no position will be more than 5% of the gross value.

To update an existing portfolio, the command might be:

```
> opt.ir3 <- trade.optimizer(prices, varian,
+    expected.return=alphas, gross.value=1.4e6,
+    net.value=c(-1e4, 2e4), ntrade=15, max.weight=.05,
+    existing=cur.port, turnover=5e4)
```

Here both the number of assets in the trade and the turnover are limited, and of course the existing portfolio is specified.

A more logical version of the command above constrains with `risk.fraction` (see page 33) rather than `max.weight`:

```
> opt.ir4 <- trade.optimizer(prices, varian,
+    expected.return=alphas, gross.value=1.4e6,
+    net.value=c(-1e4, 2e4), ntrade=15, risk.fraction=.05,
+    existing=cur.port, turnover=5e4)
```

Maximize Return with a Bound on the Variance

If you have a target for the maximum variance that you want, then you can add a constraint on the variance. Suppose that volatility should be no more than 4%. We need to transform the 4% into a variance, and (possibly) go from annual to the frequency of the variance. If the variance is on a daily frequency (and created from returns in decimal as opposed to percent), then the command would look like:

```
> opt.vc1 <- trade.optimizer(prices, varian,
+    expected.return=alphas, gross.value=1.4e6,
+    net.value=c(-1e4, 2e4), ntrade=55, risk.fraction=.05,
+    var.constraint=.04^2/252)
```

This is not doing quite what is advertised in the headline, but may actually be more appropriate. This is really maximizing the information ratio with a constraint on the variance. If the best information ratio has a variance that is smaller than the bound, then it will be the best information ratio that is chosen.

If you truly want the maximum return given the variance bound, then you need to say so explicitly:

```
> opt.vc2 <- trade.optimizer(prices, varian,
+    expected.return=alphas, gross.value=1.4e6,
+    net.value=c(-1e4, 2e4), ntrade=55, risk.fraction=.05,
+    var.constraint=.04^2/252, utility='maximum return')
```

Minimize Variance Given a Long List and a Short List

Suppose you don't have specific predictions for the returns of the assets, but you do have a set of assets that you predict will go up and a set you think will go down. That is, you have a buy-hold-sell list. In this case you can force only buys from the long list, sells from the short list, and minimize the variance of the portfolio.

If `long.names` and `short.names` are vectors containing the asset names that are predicted to go up and down, then the computation can proceed along the lines of:

```
> long.zero <- rep(0, length(long.names))
> names(long.zero) <- long.names
> short.zero <- rep(0, length(short.names))
> names(short.zero) <- short.names
```

S code note

These commands create two new vectors containing all zeros and with names that are the long assets for one of them, and names that are the short assets for the other. The `rep` function replicates its first argument.

These new vectors are then used as the `lower.trade` and `upper.trade` arguments:

```
> opt.list1 <- trade.optimizer(prices, varian,
+     gross.value=1.4e6, net.value=c(-1e4, 2e4),
+     lower.trade=long.zero, upper.trade=short.zero,
+     universe.trade=c(long.names, short.names),
+     ntrade=50)
```

The `lower.trade` and `upper.trade` arguments to `trade.optimizer` are limits on the number of units (e.g., lots or shares) that can be traded. Each asset can be given a different limit. If what is passed in is a vector with names, then any number of assets may be represented (in any order). So using `long.zero` as the `lower.trade` argument says that the assets named by `long.zero` can not have negative trades—that is, can only be bought. The same logic applies to the `upper.trade` argument with `short.zero`.

Another important piece of the command is setting `universe.trade` so that assets that are in neither `long.names` nor `short.names` won't trade at all.

This example is a simple case—the lower and upper bounds can be more complicated when there is an existing portfolio. It is possible that using the `positions` argument (page 38) would be easier in such a case.

If, for example, the existing portfolio has a long position in an asset that is on the short list, you may wish to force a sell in the asset that is at least as large as the existing position. The commands above will not do that— you need to resort to either the `forced.trade` argument or the `positions` argument.

```
> opt.list2 <- trade.optimizer(prices, varian,
+     gross.val=1.4e6, net.val=c(-1e4, 2e4),
+     lower.trade=long.zero, upper.trade=short.zero,
+     universe.trade=c(long.names, short.names),
+     ntrade=50, forced.trade=c(ABC=-25))
```

In this example we are forcing a sell of at least 25 units of asset ABC.

Mean-Variance Optimization

To do mean-variance optimization, you need to give both a variance and expected returns and set:

```
utility = "mean-variance"
```

You will want to specify the risk aversion as well. More details of the risk aversion parameter are on page 123.

```
> opt.mv1 <- trade.optimizer(prices, varian,
+    expected.return=alphas, gross.value=1.4e6,
+    net.value=c(-1e4, 2e4), ntrade=55, risk.fraction=.05,
+    utility='mean-var', risk.aversion=3.7)
```

Alternatively, you can do mean-volatility optimization by using:

```
utility = "mean-volatility"
```

6.3 Managing Cash Flow

The optimizer can be used to select trades to take money out of the portfolio, or put money into it. The optimizer makes this process quite painless, even when you want a fair amount of control over the trade.

The examples given below assume that the cash management is via the gross value. An alternative would be to control the longs and shorts directly. You would use the `long.value` and the `short.value` arguments for this more specific control.

Injecting Money into a Portfolio

If you want to perform optimization with a constraint on the variance (see page 83), then a command that will do this while only increasing existing positions is:

```
> curgross <- valuation(cur.port, prices)$total["gross"]
>
> opt.in1 <- trade.optimizer(prices, varian,
+    expected.return=alphas, gross.value=curgross + newcash,
+    net.value=c(-1e4, 2e4), ntrade=15, risk.fraction=.05,
+    var.constraint=.04^2/252, existing=cur.port,
+    turnover=newcash, universe.trade=names(cur.port))
```

The key points here are setting the gross value to the desired new amount, and forcing the turnover to equal the amount by which the gross is increased—this means that no positions can be reduced. Setting the trade universe to be the assets that are in the current portfolio ensures that no new positions are opened. If it is okay to open new positions, then do not give the `universe.trade` argument. You can also increase `turnover` any amount that you want in order to do rebalancing as well as managing cash.

Extracting Money out of a Portfolio

Taking money out of the portfolio is similar to putting it in, a suitable command might be:

```
> curgross <- valuation(cur.port, prices)$total["gross"]
>
> opt.out1 <- trade.optimizer(prices, varian,
+    expected.return=alphas, net.value=c(-1e4, 2e4),
+    gross.value=curgross - cash.required, ntrade=15,
+    turnover=cash.required, risk.fraction=.05,
+    var.constraint=.04^2/252, existing=cur.port,
+    universe.trade=names(cur.port))
```

The comments about the `universe.trade` and `turnover` arguments made about injecting money also apply here.

The formulations given for both injecting and extracting money are implicitly assuming that the optimizer will want to trade at least the amount requested. That doesn't have to be true. A safer approach would be to give `turnover` an interval, perhaps something like:

```
turnover = cash.required * c(1, 1.01)
```

6.4 Money Constraints

If you are looking for a control in the optimizer for leverage, there isn't one. The amount of leverage applied is determined by the fund manager—leverage is external to the optimization. The decision on leverage comes first, which determines the gross value and the net value—quantities that the optimizer does need to know.

caution

The expected return(s) (alpha.values) and variance(s) of the portfolio are given in terms of the assumed net asset value, which is the gross value. If your actual net asset value is different than this, then you need to adjust the results.

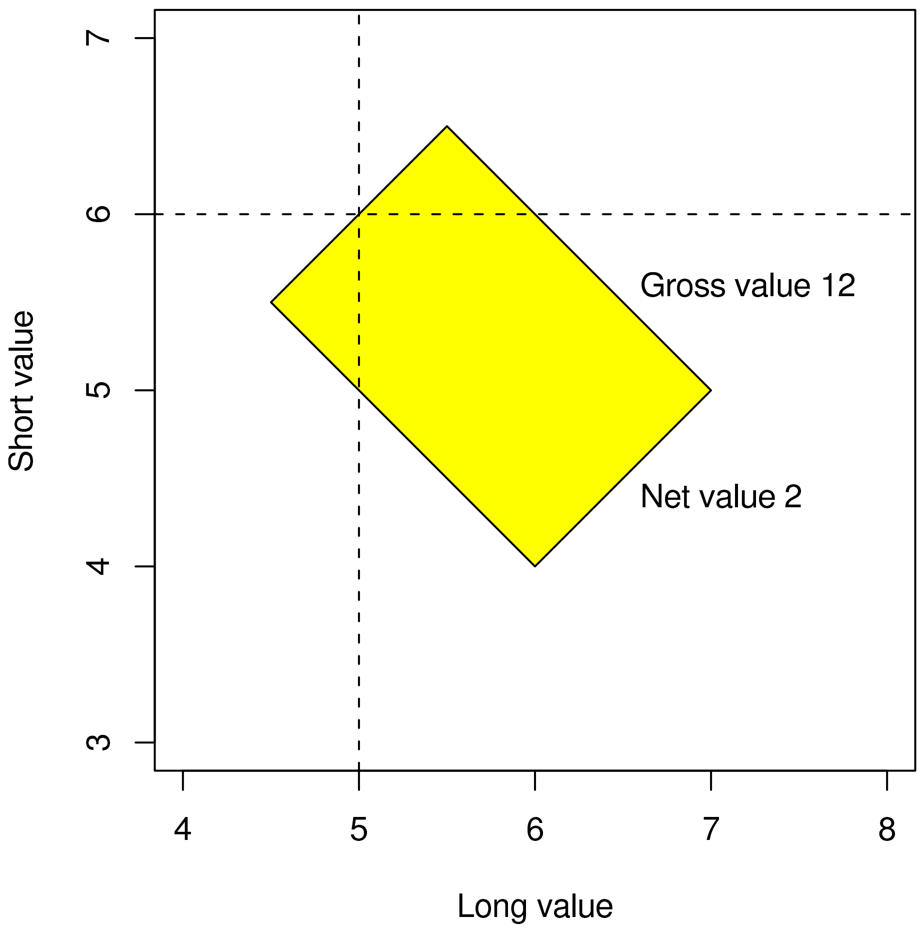
The constraints on the long and short values are not just redundant to the gross and net value constraints. Figure 6.1 shows that if we plot the gross and net value constraints, then these form a rectangle that is diagonal to the long value and short value axes. The constraints displayed in the figure are the gross value between 10 and 12, and the net value between -1 and 2. Constraints on the long value and the short value will be vertical and horizontal sections, so adding these change the shape of the allowable region.

6.5 Real-Time Monitoring

You may want to perform real-time optimization. The prices will change from minute to minute, and if your expected returns change continuously as well, then constantly updating the optimization could be valuable. Such a process is very easy to set up. Here is an example:

```
> repeat{
+   prices <- update.prices() # user-defined function
+   alphas <- update.alphas() # user-defined function
```

Figure 6.1: Constraints on gross, net, long and short values.



```
+   cur.port <- update.portfolio() # user-defined function
+   cur.opt <- trade.optimizer(prices, varian,
+       expected.return=alphas, turnover=5e4, ntrade=5,
+       existing=cur.port)
+   if(length(cur.opt$trade)) {
+       deport(cur.opt, what="trade")
+   }
+ }
```

S code note

In S, `repeat` performs an infinite loop. Generally the body of a repeat loop will contain a test of when the loop should be exited. In this case there is no such test—the optimizations will continue to be performed until you explicitly interrupt the loop. It would be easy enough to use the `date` function to check the time of day and exit the loop after the close of trading.

This example has three functions that you would need to write. These functions update the prices, the expected returns, and the positions in your portfolio. If you are willing to assume that the suggested trades always happen, then you could merely have:

```
cur.port <- cur.opt$new.portfolio
```

The technique here might be characterized as looking for the next small thing—the size of the trade and the maximum number of assets to trade should probably be small.

6.6 Going Farther

Tasks that you might want to undertake:

- To add trading costs to your optimization command, see Chapter 8 on page 95.
- To add constraints, see Chapter 3 on page 27.
- To review common mistakes, see Section 9.1 on page 101.
- To improve your solution or examine the quality of solutions you are getting, go to Chapter 11 on page 115.
- To export your solution to a file, see Section 7.3 on page 93.
- To perform optimizations with multiple variances and/or multiple expected return vectors, see Chapter 13.
- If you want a benchmark with a long-short portfolio, see Section 10.2.

Chapter 7

General Use

The core functionality of Portfolio Probe is generating random portfolios and optimizing trades. More detailed description of these activities can be found in other chapters. This chapter focuses on peripheral tasks. In particular there are hints about using the S language.

7.1 Setting Up Data

Before computations can be performed, there needs to be data in place on which to do the computations. Data that we may need include prices, existing positions in the portfolio, expected returns, variances, membership of assets in various groups such as countries and sectors, and so on.

Prices and Other Imports

You will probably need to import prices into S. You may need to get other data into S as well. A typical command to create a price vector is:

```
> prices <- drop(as.matrix(read.table("prices.txt",
+     sep="\t")))
```

or

```
> prices <- drop(as.matrix(read.table("prices.csv",
+     sep=",")))
```

depending on whether your file is tab-separated or comma-separated. (Other file formats can be handled as well.)

S code note

These commands are assuming that there are two items on each line of the file—an asset name and a number which is the price of a given quantity of the asset. The `read.table` function returns a type of S object called a *data frame*. In S data frames and matrices are both rectangles of entries. A matrix must have all of its entries the same type—for example all numbers or all character strings. A data frame has the freedom to have different types of entries in different columns—one column could be numeric and another categorical.

In this example `read.table` creates a data frame that has one column, which is numeric, and the asset names are used as the row names. This is then converted from a data frame into a matrix. Finally the `drop` function changes the object from being a one-column matrix into an ordinary vector. The reason that there is the intermediate step of converting to a matrix is that the row names are not preserved when a data frame is dropped to a vector. (There is a reason for this—you can just think of it as an unfortunate fact.)

Other data will be imported in a similar fashion. Using `as.matrix` is fairly likely, `drop` is only appropriate when there is a single column (or row) of data and a simple vector is desired.

Variance Matrix

A variance matrix is often used in Portfolio Probe. If your problem is an asset allocation with at most a few dozen assets, then a sample variance matrix as computed by the `var` or `cov.wt` S functions should do (though there may be better methods). For a large number of assets, a factor model or a shrinkage model will probably be more appropriate.

You can use any variance matrix that you want. It can come from a commercial source or you can compute it yourself.

If you need to compute the variance matrix yourself, there are functions in the `BurStFin` package that will do that.

If you are using R on Windows, then you can get `BurStFin` with:

```
> install.packages("BurStFin",
+       repos="http://www.burns-stat.com/R")
```

If you are using a recent enough version of R, then you can get it from CRAN. That is, you don't need to have the second argument in the command above.

You can get the source with:

```
> download.packages("BurStFin", 'some_directory',
+       type="source", repos="http://www.burns-stat.com/R")
```

Once the package is installed for your version of R, you can do:

```
> library(BurStFin)
```

in each session in which you want to use its functionality (this command might be slightly more complicated if you are using S-PLUS).

The `factor.model.stat` function in `BurStFin` creates a variance matrix using a factor model. The command to get a variance matrix can be as simple as:

```
> retmat <- diff(log(price.history))
> varian <- factor.model.stat(retmat)
```

Alternatively, a variance that shrinks towards the equal correlation matrix can be computed with:

```
> varian <- var.shrink.eqcor(retmat)
```

S code note

Here `price.history` is a matrix of asset prices with the columns corresponding to the assets and the rows corresponding to times (the most recent time last). The calculation produces a matrix of returns which we call `retmat` in this case. (The vector that is used as the `prices` argument could be the final row of the price history matrix.)

The `log` function returns an object that looks like its input, but each element of the output is the natural logarithm of the corresponding element of the input.

The `diff` function when given a matrix performs differences down each column. The first row of the result is the second row of the input minus the first row of the input; the second row of the result is the third row of the input minus the second row of the input; *et cetera*.

---

note

The `retmat` object holds log returns (also known as continuously compounded returns). You may hear strong opinions on whether simple returns or log returns are preferred. In mean-variance optimization neither is entirely correct, however the results are likely to be quite similar—especially if daily or higher frequency returns are used.

---

caution

Do not use daily returns when your data include assets from markets that close at different times—global data in particular. The asynchrony of the returns means that the true correlations are higher than those that will be estimated from the data. Thus optimizations will be distorted. Weekly is the highest frequency that should be used with data that are substantially asynchronous. (Note though that techniques do exist to adjust for asynchrony—see [Burns et al., 1998].)

---

caution

There is the possibility of confusion with the word "factor". In S an object that is of type "factor" is something that is categorical. For example, a vector of the sector or country of assets will (at some point at least) be a factor when used as a linear constraint. A factor model does not contain any factors in this sense.

A large amount of the effort in `factor.model.stat` is dealing with missing values. The default settings are reasonable for long-only optimizations. Possibly a more appropriate matrix for benchmark-relative analyses and long-short optimization would be:

```
> varian <- factor.model.stat(retmat, zero=TRUE)
```

For assets with missing values this biases correlations toward zero rather than towards the average correlation as the previous command does.

advanced

The treatment of missing values can have a material impact on the optimization, and it can be beneficial to specialize the estimation to your particular application. An easy way to do this is to return the factor model representation:

```
> varfac <- factor.model.stat(retmat, out="factor")
```

When the `output` argument is set to `"factor"`, the result is a list with components named `loadings`, `uniquenesses` and `sdev`. You can modify these to correspond to an approach appropriate for you. Then you can create the variance matrix from the object:

```
> varian <- fitted(varfac)
```

This works because the result of `factor.model.stat` when it returns the factor representation—`varfac` in this case—has a class, and that class has a method for the `fitted` generic function. The result of this function is the variance matrix that is represented by the factor model.

Adding a Benchmark to the Variance

Perhaps you have a variance matrix of the individual assets, but the benchmark is not included in the variance. You can add the benchmark to the variance matrix if you have the weights for the constituents of the benchmark and all of the constituents are in your variance matrix. Merely use the `bench.weights` argument:

```
benchmark="BEN", bench.weights=list(BEN=BEN.wts)
```

The `bench.weights` argument takes a list that can contain any number of benchmarks. The component names are the identities of the benchmarks, and the components are named numeric vectors that should sum to 1 (or 100). The names on the weights must be of assets that are in the variance matrix.

If there are `risk.fraction` constraints involving a benchmark, then the `bench.weights` argument is required even if the benchmark is already incorporated into the variance.

The `var.add.benchmark` function in the `BurStFin` package returns an enlarged variance with the benchmark included.

This computation (the one done via the `bench.weights` argument or the `var.add.benchmark` function) is the preferred method of introducing a benchmark into a variance matrix for optimization. See [Burns, 2003a] for a study on approaches to incorporating benchmarks into the variance matrix for optimization.

7.2 The Random Generation or Optimization

The optimization process itself is covered in other chapters. The basics are covered in chapters that depend on what you are doing:

- Random portfolio generation: Chapter 2.

- Long-only portfolio optimization (active, passive or utility-free): Chapter 5.
- Long-short portfolio optimization: Chapter 6.

Trading costs are discussed in Chapter 8, while Chapter 3 covers constraints.
Chapters 9 and 13 have additional information—Section 9.1 on page 101 is particularly recommended.

caution

If in optimization a penalty is imposed because one or more constraints are broken, then the optimizer will produce a warning. Do not ignore the warning. It could well mean that there is a mistake in the specification.

You can re-run the optimization—perhaps with an increased number of iterations—to see if the optimizer finds a solution that does not break any constraints. If that doesn't work, you should probably investigate if you have imposed constraints that don't allow a solution. Section 9.1 on page 101 lists some common mistakes.

On the other hand, this method of treating constraints allows backtests to proceed even when the constraints that are imposed are infeasible for some of the time periods. The answers that break the constraints, while not optimal, may be (at least sometimes) reasonable.

7.3 Post-Optimization

This section is specific to optimization. If you are generating random portfolios, then the analogous discussion is Section 2.2.

Once you have performed an optimization, there are two tasks remaining—see what the trade is, and export the data so that the trade can be implemented.

Explore the Trade

Information on the optimization can be viewed using the `summary` function:

```
> opti <- trade.optimizer(prices, varian, long.only=TRUE,
+    gross.val=1e6)
> summary(opti)
```

The `summary` function provides information on:

- the utility, costs, and penalties for broken constraints
- opening and closing positions, etc.
- valuations of the portfolio and trade
- realization of linear (and count) constraints
- realization of risk fraction constraints

- realization of distance constraints

You can get some more specific information by printing the optimal portfolio object (which can be done by just typing its name):

```
> opti
```

Alternatively, you could look at an individual component of the object, such as the trade:

```
> opti$trade
```

The prices that `summary` uses to value the portfolio are, by default, the prices used in the optimization. You can give `summary` different prices:

```
> summary(opti, price=new.prices)
```

Export the Trade

The other task to be performed—once you are satisfied with the optimization—is to place the information elsewhere. The `deport` function does this ("export" is a logical name for this functionality, but that has a different meaning in R). The simplest use of this function is just to give the name of the portfolio object:

```
> deport(opti)
[1] "opti.csv"
```

The return value of the function is a character string giving the name of the file that was created. If the optimization was performed in terms of lots but you want the file to reflect number of shares, the `mult` argument should be given. If, for instance, all of the lot sizes are 100, then you would do:

```
> deport(opti, mult=100)
[1] "opti.csv"
```

When there are different lot sizes, you should give a vector containing the lot size of each asset:

```
> deport(opti, mult=lotsizes)
[1] "opti.csv"
```

If you want the file to be in monetary units, then the `mult` argument should be prices:

```
> deport(opti, mult=prices, to="txt")
[1] "opti.txt"
```

This command writes a tab-separated file (which in this case is called `opti.txt`). The `what` argument can be given to control what information is put into the file—see the help file for details.

7.4 Going Farther

- See Chapter 4 for the valuation (and returns) of portfolios.

Chapter 8

Trading Costs

This chapter covers trading costs. It is possible to constrain costs, so costs can be used when generating random portfolios. This is only likely when imitating a series of trades with random trades.

Costs are a tricky part of optimization, but are extremely important.

8.1 Background

If there were any justification for the attitude that optimization is hard, it should be on account of trading costs. However, the problem of trading costs is fundamental to fund management—you have to confront the issue whether or not you use an optimizer.

There are numerous sources of trading costs—commissions, bid-ask spread, and market impact are commonly considered. Less obvious costs can be such things as capital gains taxes if a position is sold, and the cost of imperfectly known expected returns. Costs need not be symmetric for buying and selling. You may, for example, want the cost of increasing positions to be more than the cost of decreasing them, because of increased liquidity risk and/or because of the implicit cost of needing to close the position again.

There are (at least) three key questions with transaction costs:

- What are the costs (both visible and invisible)?
- How should those costs be amortized?
- How should the amortized costs be added to the utility?

The first question is fundamental to asset management.

The second and third questions are challenging as well, and discussed more fully in Section 8.4. Once the last question is answered, the `ucost` argument allows an easy way to implement the answer. The cost for a trade is multiplied by `ucost` before it goes into the utility computation.

8.2 Specifying Costs

Costs are specified similarly in `trade.optimizer` and `random.portfolio`. When generating random portfolios, costs will be ignored unless `limit.cost` is given (or the utility is constrained).

The specification of costs is very flexible while simple costs can be given easily. To allow for asymmetry in costs, there are four cost arguments:

- `long.buy.cost`
- `long.sell.cost`
- `short.buy.cost`
- `short.sell.cost`

This allows differences in the costs for long positions versus short positions, as well as differences for selling and buying.

If there is no difference between buying and selling, and between long and short positions, then give only the `long.buy.cost` argument.

caution

If you give one of the cost arguments other than `long.buy.cost` but no others, then the named situation will be the only case where there is cost. That seems unlikely to be what you want.

Minimally, costs must be given for all of the assets that are allowed to trade in the problem (if costs are given at all). Costs may be given for any number of assets as long as all the tradeable assets are included.

Linear Costs

Linear costs are given with a vector (or a one-column matrix) of costs. For example, to have costs of 30 basis points for all trades, the command would be:

```
> opt1 <- trade.optimizer(prices, ...,
+    long.buy.cost = prices * .003)
```

To have costs of 50 basis points for short sales and costs of 30 basis points for all other trades, the command is:

```
> opt2 <- trade.optimizer(prices, ...,
+    long.buy.cost = prices * .003,
+    short.sell.cost = prices * .005)
```

The "..." in these commands and subsequent ones refers to additional arguments that you want in the optimization.

The advantage of linear costs is that they are easy to think about and easy to write down. Linear costs are undoubtedly more popular than they otherwise would be because computational engines have generally been unavailable for more realistic models of cost.

Nonlinear Costs

Costs are unlikely to really be linear. It can be quite useful to have non-integer exponents in the trading cost functions. This is allowed via the `cost.par` argument. `cost.par` is a vector giving the exponents for the cost function.

Consider:

```
> cost.poly
    [,1] [,2]  [,3] [,4]
ABC 0.00 0.01  0.02 0.00
DEF 0.03 0.02 -0.01 0.01
> trade.optimizer( ..., long.buy.cost=cost.poly)
> trade.optimizer( ..., long.buy.cost=cost.poly,
+    cost.par=0:3)
```

These two calls to `trade.optimizer` are equivalent (though there would be a slight computational efficiency advantage to the first one). Polynomial costs are easy to specify, but unlikely to be realistic.

A more telling use of `cost.par` is:

```
> trade.optimizer( ..., long.buy.cost=cost.poly,
+    cost.par=c(0, 1, 1.5, 2.34))
```

In this case if `ABC` trades 30 lots and `DEF` trades -25 lots, then the cost for `ABC` will be:

$0 + .01(30) + .02(30^{1.5}) = 3.586$

The cost for `DEF` will be:

$.03 + .02(25) - .01(25^{1.5}) + .01(25^{2.34}) = 17.952$

When `cost.par` is used, there is a restriction that all of the cost matrices must have the same number of columns as `cost.par`. In actuality there is no loss of generality as columns of zeros can be added to matrices as required, and `cost.par` can be the union of all of the desired exponents.

We've seen `cost.par` as a vector of exponents. It is also possible to give `cost.par` as a matrix with rows corresponding to assets. This allows each asset to have its own set of exponents as well as coefficients.

If `cost.par` were:

```
    [,1] [,2] [,3] [,4]
ABC    0    1  1.5 2.34
DEF    0    1  1.5 2.34
```

then it would yield just the same as before. But the point of giving a matrix is that not all rows have to be the same:

```
    [,1]  [,2]  [,3]  [,4]
ABC    0     1  1.57  2.11
DEF    0     1  1.45  2.62
```

8.3 Power Laws

[Grinold and Kahn, 2000] present an argument that the trading cost per unit of an asset should be commission plus half the bid-ask spread plus some constant times the square root of the quantity: the amount to be traded divided by the average daily volume. The last term involving the square root comes from an inventory model. They suggest that the constant should be on the order of the daily volatility of the asset.

Sample code implementing this model might look like:

```
> cost.mat <- cbind(commission + spread[names(prices)] / 2,
+    volatility[names(prices)] / sqrt(ave.daily.volume[
+    names(prices)]))
>
> trade.optimizer(..., long.buy.cost=cost.mat,
+    cost.par=c(1, 1.5))
```

The first command creates a two-column matrix. The first column relates to the items that are linear (commission and spread), the second column contains the cost involving the square root of the trade volume. When this is used in an optimization, the exponents for these columns are 1 and 1.5, respectively.

The second coefficient is 1.5 (where naively you may expect it to be 0.5) because the square root rule says that the cost *per share* depends on the square root of the number of shares traded. (If you think in terms of calculus, we are taking the integral of the cost function.)

[Almgren et al., 2005] empirically found a power of 0.6 to fit a set of trades on large US stocks. Since a square root is a power of 0.5, their findings suggest that trading costs grow slightly faster as the size of the trade increases.

If you were to use this power law, then the `cost.par` in the example above would be 1 and 1.6 instead of 1 and 1.5. Except maybe not. Use 1.6 if the coefficient you have is *per share*. You would use 0.6 if the coefficient you have is for the market impact of the trade with the linear costs stripped out.

8.4 On Scaling Costs Relative to Utility

There are (at least) three things we want in the costs that we declare:

- The cost of a larger trade in an asset is appropriate relative to a smaller trade of that asset.
- The cost of trading one asset is well-scaled relative to the cost of trading the other assets.
- The costs are appropriately scaled in light of the utility that is used.

The proper scaling of the costs relative to the utility—whether that is expected returns (and variance), or distance—is the point where the science gets a bit fuzzy. This is also a key issue controlling the merit of the optimization. If the costs given the optimizer are too small, then excessive trading eats up performance. If the given costs are too large, then performance is hurt via lost opportunity. Perfection is not necessary for the optimization to be of value, but attempting perfection is probably not such a bad mission.

The added utility of a trade needs to compensate for the transaction cost. Thus the costs should correspond to the expected returns over the expected holding period.

To be more specific, suppose that we think it will cost 50 basis points to trade a certain asset. If we are using daily data and our expected return for this asset is 10 basis points, then we will have to have a holding period of 5 days in order to break even. To amortize the cost, it (that is, the 50 basis points) should be divided by the number of days that we expect to hold this asset. If we expect to hold it for 2 days, the optimizer would see a cost of 25 basis points. If we expect to hold it 10 days, the optimizer would see a cost of 5 basis points. (The statement of this example might seem to imply that volatility is of no concern—it is of concern.)

Let's revisit the example from Section 8.3:

```
> cost.mat <- cbind(commission[names(prices)] +
+    spread[names(prices)] / 2,
+    volatility[names(prices)] / sqrt(ave.daily.volume[
+    names(prices)]))
>
> cost.amort <- cost.mat / expected.holding[names(prices)]
>
> trade.optimizer(..., long.buy.cost=cost.amort,
+    cost.par=c(1, 1.5))
```

This adds a command which divides (what might be called) the raw costs by how long we expect to hold each asset. The result is the amortized costs. The holding period should be in the same time units as the expected returns. If we have daily returns, then an asset that we expect to hold 25 days should have 25 as its expected holding period. However that same asset should have an expected holding period of about 0.1 when we are using annualized returns.

You can use the `ucost` argument to scale the cost relative to the other items in the utility.

That the expected returns are noisy suggests that the costs should be increased from this "rational" value to reduce trading.

There is another sense of scaling costs. By default `trade.optimizer` sums the costs from all of the trading and then divides by the gross value of the portfolio. This is the natural thing to do as the expected returns and the variance are also scaled by the gross value when the utility is computed. However, you do have the option to scale by the trade value or not to scale the costs at all—this is controlled by the `scale.cost` argument.

8.5 Costs Due to Taxes

Taxes present a cost structure that is entirely different from trading costs resulting from the market. In particular the cost will be different depending on whether you are buying or selling. More detail on costs due to taxes can be found in [diBartolomeo, 2003] and its references.

Several optimizers implement piecewise linear cost functions. While this choice is likely to be due to mathematical tractability, costs (and benefits) from taxes really are piecewise linear in some circumstances. The tax liability often depends on the length of time an asset has been held, and there may be several holding periods for a particular asset. Portfolio Probe does not have piecewise linear costs, but you can approximate them with nonlinear costs.

Taxes also have a sort of time decay similar to options. Their effect changes as the end of the tax period gets closer.

8.6 Going Farther

Additional steps might be:

- To look for trouble spots with Chapter 9 on the next page

Chapter 9

Practicalities and Troubleshooting

The best question to ask after a computation has been performed is: **Does it make sense?**

The purpose of this chapter is to help with that question—to give hints about how it might not make sense, and to give possible solutions when it doesn't.

Though there will be a tendency to come to this chapter only when you think something is wrong, the best time to come to it is when you think everything is right.

While some of the problems listed in this chapter are specific to optimization, there are many that will affect the generation of random portfolios also.

The sections of this chapter are:

1. Easy ways to be wrong.
2. Suppressing warnings (page 104).
3. Cheatsheets (page 107).
4. Troubleshooting (page 109).
5. S Language Problems and Solutions (page 110).

9.1 Easy Ways to Be Wrong

The phrase "garbage in, garbage out" has gone slightly out of fashion, but it still has currency. What follows are a few possibilities for garbage.

Data Mangling

The prices are not all in the same currency.

It doesn't matter which currency is used, but there needs to be only one.

There are prices for the wrong quantity.

For example, the optimization is thought of in terms of lots, but the prices for some or all of the assets are for shares.

There are stock splits, rights issues, etc. that have not been accounted for.

Stock splits and rights issues that are not caught can mean that your existing holdings are wrong in the optimization.

A missed split in the price history degrades variance estimates that are based on returns created from the price history.

The variance is of prices, not returns.

The variance matrix of the returns of the assets is needed. If the variance is of the prices, the results will be seriously wrong. (The functions in `BurStFin` that create variance matrices from data give a warning if prices are used.)

The variance and/or expected returns are not properly scaled.

The scaling that is chosen does not matter, but it needs to be consistent. The (time) scales of the variance and the expected returns should match—for example they can both be annualized, or both be daily. You also need such things as benchmark constraints to be scaled to match the variance. Costs need to account for the time scale of the expected returns, and whether or not the expected returns are in percent.

The variance matrix is singular.

While the Portfolio Probe optimization algorithm will happily handle singular matrices, the results need not be useful. If there are more assets than time points in the returns data, then a factor model or shrinkage estimator rather than a sample variance should be used. This problem is certainly more serious for long-short portfolios, but is problematic for long-only portfolios as well.

If the true variance really is singular—for example cash is one of the assets (see page 138)—then you should get good results. The problem is if the variance matrix says that there are portfolios that are essentially riskless when they are not. That is, you want the variance to be decidedly positive definite excluding benchmarks and cash.

The variance matrix is not symmetric.

The optimizers inherently assume that the variance is symmetric. They will get confused if this is not the case. If what you are using as the variance is not symmetric, then you can get a version that is with:

```
> varian.sym <- (varian + t(varian)) / 2
```

There is asynchrony in the data.

Global data should not be used at a higher frequency than weekly unless asynchrony adjustments are made. In addition to different opening hours, asynchrony can be caused by assets that do not trade often. The illiquid assets will have stale prices. A model that adjusts for asynchrony due to different opening hours is presented in [Burns et al., 1998].

The expected return for a benchmark is not equal to the weighted average of the expected returns of its constituents.

This problem is pointed out in [Michaud, 1998]. Minor violations are unlikely to have much affect, but differences do distort the optimization. This is not an issue if you use the `bench.weights` argument and do not include benchmarks in the expected returns.

The signs of covariances in the variance matrix are reversed for assets that are intended to be short.

While this could produce valid results if properly done, it is confusing and totally unnecessary. Portfolio Probe was developed with long-short portfolios in mind—bizarre manipulations can be eliminated.

Input Mangling

While many mistakes are caught by the software, it is quite possible to confuse the inputs. One check is to see if the optimizer is using the objective that you intend. You can do this by:

```
> opt.1$objective.utility
[1] "mean-variance, risk aversion: 0.02"
```

The name of an argument is inadvertently dropped.

For example, you add a starting solution to a command, but forget to name it `start.sol` so the optimizer thinks it is a different argument. Most arguments are checked for validity, but in some cases the checks may not be good enough.

You confuse the meaning of “start.sol” with “existing”.

The `start.sol` argument should be one or more solutions that are used as starting values for the trade in optimization. The current portfolio should be given in the `existing` argument.

You confuse the meaning of the “lower.trade” or “upper.trade” argument.

If you habitually abbreviate the `lower.trade` argument to `lower`, you may begin to think of the bound as being on the final portfolio rather than on the trade.

Bounds for linear constraints are not properly scaled.

Values in `lin.bounds` need to match the corresponding value in `lin.style`. If the bounds are in values but the style is in weight, then there probably isn't a real constraint. Alternatively if the bounds are in weights and the style is in value, then the constraint is likely to be impossible.

You do not give the lin.abs or lin.trade argument when needed, or reverse the meaning.

If the bounds are set for net constraints but `lin.abs` is not given, then the constraints could be inconsistent. Even worse and more likely, they may not be inconsistent and give an answer that on the surface seems okay.

You give just one trading cost argument, but it is the wrong argument.

If you want costs to be the same whether the position is long or short, and whether you are buying or selling, you need to give the `long.buy.cost` argument. If you give a different argument, then the costs correspond to the named situation only and the costs in other situations are zero. There will be a warning if you do this.

You maximize the information ratio and the portfolio can reproduce the benchmark.

In this case the portfolio variance goes toward zero so the information ratio goes toward infinity. The real problem is probably that you want to have constraints so that the portfolio can not look so much like the benchmark. One workaround is to put a lower limit on the variance using the `var.constraint` argument.

9.2 Suppressing Warning Messages

While warning messages help avoid mistakes, they can be annoying (and counterproductive) if they warn about something that you know you are doing. If you have warning messages that you are expecting, it is easy to overlook warnings that you are not expecting and that portend danger.

The `do.warn` argument of `random.portfolio` and `trade.optimizer` can be used to suppress most warnings. Here is an example of its use:

```
do.warn = c(utility.switch=FALSE, value.range=FALSE)
```

Note that abbreviation of the names is allowed as long as the abbreviation is unique among the choices.

Some warnings are never suppressed—these are more likely to be mistakes. You should reformulate the command to avoid such warnings whether or not the original is a mistake.

The default is `TRUE` for all entries except `converged` (which defaults to `FALSE` since non-convergence of the optimization is seldom of concern).

Here are the possible entries for `do.warn`:

- `alpha.benchmark`: When `bench.weights` is given, the expected return for each benchmark is computed as the weighted sum of the asset expected returns. If the expected return for the benchmark is given and is different than the computed value, then a warning is issued (and the given value overrides the computed value).

- `back.compat`: The warnings, if any, that apply are different from version to version. The warnings are of changes between versions of Portfolio Probe that may be significant.

- `benchmark.long.short`: A benchmark is used in a long-short portfolio. While this can be correct, it often is not. See Section 10.2.

- `bounds.missing`: There are rows missing from the `lin.bounds` object that should logically be there. The missing bounds are taken to be infinite. This is almost certainly caused by something going wrong, or at least sloppy use.

- `converged`: (optimization only) Convergence is declared if more than `fail.iter` consecutive iterations fail to improve the solution. If the maximum number of iterations is performed before this condition occurs, then there is no convergence for the run. When `stringency` is positive, the convergence is a little more complicated.

- `cost.intercept.nonzero`: One or more of the cost arguments have non-zero values for the intercept. (This only applies when polynomial costs are given.) This may well be correct, but it is easy to forget the intercept column in the cost matrix.

- `dist.prices`: One or more of the assets that are in `prices` are not in (a component of) `dist.prices` and hence are assumed to be zero.

- `dist.style`: Since it is easy to make a mistake between giving shares or values to the `dist.center` argument, a check is made to see if what is stated in `dist.style` fits into the gross value range that is given or inferred.

- `dist.zero`: One or more of the assets in (a component of) `dist.center` have a non-zero value but a price of zero in `dist.prices`.

- `exit.obj`: The optimization was exited early due to the objective being better than the `exit.obj` control value.

- `extraneous.assets`: One or more objects contain data on assets that are not in `prices` and hence not of use in the computation.

- `index.zero`: An argument that expects zero-based numbers is given and it is ambiguous whether the values are zero-based or one-based.

- `ignore.max.weight`: One or more assets in the existing portfolio break their maximum weight constraint; and either maximum weights are not forced to be obeyed, or the trade can not be forced to be large enough. See page 32.

- `infinite.bounds`: All bounds in `lin.bounds` are infinite, meaning there are no bounds at all.
- `max.weight.restrictive`: The vector of maximum weights is very restrictive. That is, the sum of (a subset of) the maximum weights is only slightly more than 1.
- `neg.dest.wt`: (optimization only) There is at least one negative destination weight in the utility table. This is almost surely wrong unless the problem is outside typical portfolio optimization.
- `neg.risk.aversion`: (optimization only) There is at least one negative risk aversion value. Risk aversion is traditionally thought to be non-negative, but you could have a special situation.
- `no.asset.names`: One or more objects do not have asset names, and hence require that they be in the same order as the assets in `prices`. Some objects are allowed to not have asset names in order to facilitate quick experiments—asset names are always recommended for production situations.
- `noninteger.forced`: There is a forced trade that is constrained such that the trade must be non-integer. This could be something you want to do, but can mean that something is wrong with the specification of constraints.
- `nonzero.penalty`: (optimization only) The penalty is non-zero, meaning that not all constraints are satisfied.
- `notrade`: The formulation does not allow any trading. If you really want no trading, then the recommended approach is to set `ntrade = 0`.
- `novariance.optim`: (optimization only) An optimization (other than with a distance) is requested but no variance is given. This can be a reasonable operation for some long-short situations, but is generally suspicious.
- `penalty.size`: (optimization only) The risk aversion parameter is large relative to the values in `penalty.constraint`. In this case the optimizer may have problems getting the constraints to be obeyed.
- `positions.names`: The asset names in the `positions` argument are not the same as the universe of assets in the problem.
- `random.start`: (random portfolios only) The `start.sol` argument is given, which is ignored in `random.portfolio`. `start.sol` is sometimes confused with `existing` (the current portfolio).
- `randport.failure`: (random portfolios only) Fewer than the requested number of random portfolios were generated.
- `riskfrac.part`: The `risk.fraction` argument only applies to one variance-benchmark combination but there are additional combinations.
- `start.noexist`: The `start.sol` argument is given, but `existing` is not. This is sometimes what is wanted, but it might indicate that the meaning of `start.sol` is confused with `existing`.

- `superfluous.constraint`: A constraint is stated that is always satisfied.
- `thresh.notrade`: Some assets are not able to trade because their thesholds are too large.
- `turnover.max`: The maximum `turnover` is too large to have an effect. If you want to specify a minimum turnover but no maximum turnover, then this warning is avoided by setting the maximum to `Inf`.
- `utility.switch`: (optimization only) The utility (either given or default) is switched to another utility.
- `value.range`: One or more of the ranges for monetary value (`gross.value` etc.) is narrow relative to the prices.
- `var.eps`: (optimization only) If the portfolio variance goes very small when the information ratio is being maximized, then weird things can start to happen. A leading case of this is when a benchmark is used and the benchmark can be reproduced almost exactly. A fairly naive test is used to try to highlight such cases.
- `variance.benchmark`: When `bench.weights` is given, variance values are computed for each benchmark based on the weight vector. If the variance already includes the benchmark and values are different than the computed values, then a warning is issued (and the given values override the computed values).
- `variance.list`: A compact variance is given, which means that there is no way to check that the assets are the same (and in the same order) as the assets in `prices`.
- `zero.iterations`: The `iterations.max` argument can be set to zero, but this is not the same as doing no optimization. To get the `start.sol` trade as the final trade, set `funeval.max` to zero or one.
- `zero.variance`: If more than one value on the diagonal of a variance is zero (think "cash"), then it seems like trouble should happen. However, no such troubles have been observed so far.

9.3 Cheatsheets

Implied Ranges

Table 9.1 lists the true meaning of a single number for those arguments that represent a range of values. Note that `close.number` is the outlier in that its single value means exactly that value.

Threshold Inputs

Table 9.2 shows the meaning of the `threshold` argument when given a matrix with each of the allowable number of columns—a plain vector is the same as a one-column matrix. The subscript of the values in the table indicate the column

Table 9.1: Implied ranges of arguments.

argument	**input**	**meaning**
`gross.value`	x	[x * `allowance`, x]
`net.value`	see Section 3.2	
`long.value`	x	[x * `allowance`, x]
`short.value`	x	[x * `allowance`, x]
`turnover`	x	[0, x]
`ntrade`	n	[0, n]
`port.size`	n	[1, n]
`alpha.constraint`	x	[x, `Inf`)
`var.constraint`	x	[0, x]
`bench.constraint`	x	[0, x]
`limit.cost`	(must give the range)	
`close.number`	n	[n, n]

Table 9.2: The meaning of `threshold` inputs.

threshold constraint	1 column	2 columns	3 columns	4 columns
trade sell	$-x_1$	x_1	x_1	x_1
trade buy	x_1	x_2	x_2	x_2
portfolio short	0	0	NA	x_3
portfolio long	0	0	x_3	x_4

of the value. So x_1 is from the first column and x_2 is from the second column, and so on. A three-column matrix can only be given in the case of a long-only portfolio.

Positions Inputs

Table 9.3 indicates the allowable number of columns for objects given as the `positions` argument, and the meaning of those columns. See Section 3.7 for more details.

Table 9.3: The column order for the `positions` argument.

meaning	**2 columns**	**4 columns**	**8 columns**
min portfolio value	x	x	x
max portfolio value	x	x	x
min trade value		x	x
max trade value		x	x
trade sell threshold value			x
trade buy threshold value			x
portfolio short threshold value			x
portfolio long threshold value			x

9.4 Troubleshooting

This section lists a number of problems and possible solutions.

Utility Problems

The utility stays the same when I change the risk aversion.

There are at least two possibilities:

1. The objective is neither mean-variance nor mean-volatility—it could be maximizing the information ratio, minimizing variance or maximizing return. Check the `objective.utility` component of the output.
2. You are passing in a value for the `utable` argument. In this case you need to set `force.risk.aver` to `TRUE` in order for the `risk.aversion` argument to override the value that is already in the utility table.

I'm minimizing the variance, and the utility is exactly zero.

The problem is almost certainly that the objective is to maximize the information ratio. You can explicitly set the objective.

This does not occur if neither the `expected.return` nor `utable` arguments are given. It can happen if you give a vector of expected returns that are all zero.

Portfolio Problems

The optimizer is suggesting trades in which some of the assets trade a trivial amount.

Use the `threshold` argument— see Section 3.5.

The optimal portfolio has positions that are very small.

Use the `threshold` argument— see Section 3.5.

I'm building a portfolio and the optimizer is having problems getting the monetary value of the portfolio right.

One possibility is that the `max.weight` (or `risk.fraction`) argument is too small. If there are only a few assets allowed in the portfolio, the sum of allowable maximum weights can be less than 1. The error of `max.weight` being too small will be caught in simple cases. However, it is possible for the test to be fooled by a combination of restrictions on trading.

Another possibility is that the ranges of the money values (`gross.value`, etc.) are too narrow.

The optimizer is not performing a trade that I am insisting upon via the lower.trade or upper.trade argument to trade.optimizer.

Use the `forced.trade` argument—see Section 3.6.

What if I don't have prices?

You can do an asset allocation and effectively get weights back. See page 79.

I'm trying to generate random portfolios with constraints I know work, but it fails.

You can set the S language seed to some number and try the call to `random.portfolio` again. For example:

```
> set.seed(123)
```

9.5 S Language Problems and Solutions

This section provides a little help with some of the aspects of the S language that you might be most likely to bang your head against.

"Some hints for the R beginner" on the Tutorials page of the Burns Statistics website might also be of help.

Creating Matrices

There are a number of places in this document where we create a matrix. This process generally involves either `cbind` or `rbind`.

`cbind` as in "column bind" and `rbind` as in "row bind". That sounds simple enough—and it is as long as you pay attention to the details:

```
> cbind(A=c(1, 2))
     A
[1,] 1
[2,] 2
> cbind(A=1, 2)
     A
[1,] 1 2
```

Both of these forms can be useful, but they give matrices that are oriented in different directions.

Debugging

[Burns, 2011] contains a slightly expanded introduction to debugging, and there are a number of additional sources. Here is something to get you started.

Suppose we do something and get an error:

```
> update(soft.op1, expected.return=new.alpha)
Error in trade.optimizer(prices = priceten, variance = varten:
        Object "new.alpha" not found
```

Now in this instance it is clear that the problem is the `new.alpha` that we tried to use. But the error could be very much more obscure—it could be unable to find an object that we've never heard of, and possibly in a function that we didn't know we were using.

You can always do the following to get a sense of where the problem happened:

```
> traceback()
6: trade.optimizer.pre(prices = priceten, variance = varten, ...
5: trade.optimizer(prices = priceten, variance = varten, ...
4: eval(expr, envir, enclos)
3: eval(call, parent.frame())
2: update.default(soft.op1, expected.return = new.alpha)
1: update(soft.op1, expected.return = new.alpha)
```

Sometimes just seeing the traceback can be enough to understand what went wrong. Other times, no. For those other cases we can actually go into any of those function calls and see the state at the time of the error. But we can only do that if the ground was prepared before the error happened. Generally it is easy enough to reproduce the error.

We want to set the `error` option to save the contents of the function calls and not just what the calls were:

```
> options(error=dump.frames)
> update(soft.op1, expected.return=new.alpha)
Error in trade.optimizer(prices = priceten, variance = varten:
         Object "new.alpha" not found
> debugger()
Message:  Error in trade.optimizer(prices = priceten,   :
        Object "new.alpha" not found
Available environments had calls:
1: update(soft.op1, expected.return = new.alpha)
2: update.default(soft.op1, expected.return = new.alpha)
3: eval(call, parent.frame())
4: eval(expr, envir, enclos)
5: trade.optimizer(prices = priceten, variance = varten
6: trade.optimizer.pre(prices = priceten, variance = varten,
Enter an environment number, or 0 to exit  Selection: 6
Browsing in the environment with call:
   trade.optimizer.pre(prices = priceten, variance = varten
Called from: debugger.look(ind)
Browse[1]>
```

Here we have selected frame 6 (the call to `trade.optimizer.pre`) and at this point the objects that that function has created are available to us to explore. You can do:

```
Browse[1]> ls()
```

to see what objects are there. When you want to exit, just type `c` (as in continue):

```
Browse[1]> c
```

This particular error can have a few causes:

1. The object truly does not exist.
2. You mistyped the object name.
3. The object exists but in a place that is not being searched.

There are `envir` arguments to a few functions that may sometimes solve problem 3 in R.

Chapter 10

Special Instructions

This chapter contains instructions for how to overcome specific problems. For example, how to do something that the program perceives as an error. In general, these are longer versions of error or warning messages that you might receive.

10.1 Special Instruction 1: Long-only when shorts exist

It is an error to declare the portfolio to be long-only when there are short positions in the existing portfolio. If you have short positions currently and you want to create a long-only portfolio, then you should do the following:

- Declare `long.only=FALSE`
- Use `positions` to set the long-only constraint for the portfolio.

For the last item you could alternatively use `forced.trade` and `lower.trade`, but `positions` is the easier approach.

The positions argument could be built like:

```
> posarg <- cbind(rep(0, length(prices)), Inf)
> dimnames(posarg) <- list(names(prices), NULL)
```

Then used like:

```
positions = posarg
```

10.2 Special Instruction 2: Benchmark in long-short optimization

Naively using a benchmark with a long-short portfolio is probably not what you want to do. Creating a benchmark in Portfolio Probe (and elsewhere) is equivalent to being short the benchmark in the amount of the gross value.

If you have a portfolio in the genre of 120/20, then a benchmark is conceptually fine but operationally wrong. If you use the benchmark argument, you will be short 140% of the benchmark with a 120/20 portfolio while you want to be short 100%.

Whether you have a benchmark in the traditional sense or in the sense that you have a set of liabilities, there are at least two choices:

- Put the portfolio of liabilities as a short asset in the portfolio and stop it from being traded.
- Use a variance matrix that is relative to the liabilities.

There are two functions in the `BurStFin` package that may be of use in this situation. The `var.add.benchmark` function allows you to add a portfolio of assets to a variance matrix. `var.relative.benchmark` transforms a variance matrix to be relative to one of its assets.

Chapter 11

Adjusting Optimization Speed and Quality

The default control settings for the optimizer are a compromise between the time it takes to finish and the quality of the solution.

Optimization is usually done in one of two settings:

- Backtesting strategies
- Production

In backtesting, the quality of the solution is much less of an issue than the speed of the optimization. If the default setting seems too slow to you, you can decrease the maximum number of iterations allowed. Setting

```
    iterations.max = 5
```

is a possibility—this is not unreasonably small for a lot of problems.

For production where time is not critical and quality is more of concern, then an easy way to get better solutions is to add the argument:

```
    stringency = 1
```

This will take a few times longer to execute, but will give at least as good of a solution.

The rest of the chapter goes into more detail about the options that you have and what they mean.

11.1 Staying at a Given Solution

A special type of "optimization" is to just return an object where the solution is the same as the starting value. At first glance this seems like a useless operation, but in fact there are many reasons to do this—here are a few:

- See the cost of a solution

- See constraint violations of a solution
- Get expected returns, variances and so on of random portfolios (this is what `randport.eval` does)
- Ensure that another program is doing the same problem
- Examine the utility of your current portfolio, or of a proposed trade

`trade.optimizer` makes this easy. Give the trade that you want to test as the `start.sol` argument, and set `funeval.max` to zero or one. An asset allocation example is:

```
> aa.gs1 <- trade.optimizer(aprice, avar.medium,
+     aret.medium, gross.value=ten.thou,
+     long.only=TRUE, utility="mean-variance",
+     risk.aversion=.05, start.sol=e60b40, funeval=0)
> valuation(aa.gs1)$weight
   equities      bonds
        0.6        0.4
> aa.gs1$utility.value
[1] 2.08
> aa.gs1$alpha.value
[1] 6.4
> aa.gs1$var.value
[1] 169.6
```

Note that setting:

```
iterations.max = 0
```

is not the same. This still allows the pre-iteration and post-iteration operations and the result could be substantially different from the starting solution.

If the existing portfolio is where you want to stay, then you can use the argument:

```
ntrade = 0
```

11.2 Reducing Time Use

You may want to reduce the time that an optimization takes because you are doing a large number of optimizations. The only viable way to reduce time is to reduce the `iterations.max` argument.

For a lot of problems 5 iterations is likely to optimize enough that backtests will give the right indication. Zero iterations will almost surely be too few, but even one iteration may not be so terrible.

11.3 The Optimization Process

This section briefly describes the optimization algorithm so that the suggestions for improving quality in the next section will make sense.

The optimization process can be broken into three levels:

- sub-algorithms
- iterations
- runs

The algorithm for `trade.optimizer` is generally described as a genetic algorithm. That is a simplification. Each iteration performs several sub-algorithms—some of them genetic (plus simulated annealing), and others can be described as greedy algorithms.

A run is a series of iterations. The run ends when too many consecutive iterations fail to find an improvement. That number of failures is controlled by the `fail.iter` argument, which has a default of zero. So, by default, any iteration that fails to improve will cause the run to end. If `fail.iter` is 1, then it takes two consecutive iteration failures to end the run.

The default is to perform a single run. Multiple runs are done if the `stringency` argument is positive. The stringency criterion is satisfied if the best trade has been found again (in different runs) `stringency` times.

For example, suppose `stringency` is 1. After each run we have the best found so far. If the result of the new run is the same (the same assets traded, the same number of units traded) as the best found previously, then the stringency criterion is met and the process stops. If `stringency` were 2, then it would take three runs with the same result to end the process. The best runs need not be found consecutively.

When stringency is positive, there are three types of runs:

- initial runs
- final runs
- non-convergence run

Initial runs perform the optimization as stated. The number of initial runs is determined by the `runs.init` control argument.

If the stringency hasn't been satisfied in the initial phase, then the final runs are started. The key difference is that only assets that were involved in trades in the initial phase are eligible to trade in this phase. The number of final runs is controlled by `runs.final`.

If the stringency is still not satisfied at the end of the final runs, then one more run is done. This run starts with the best solution found in any of the previous runs. The restriction to trading as in the final runs remains in place. The maximum number of iterations allowed in this run is a multiple of the maximum number of iterations for the other runs—that multiple is given by the `nonconverge.mult` control argument.

11.4 Improving Quality

The easiest—and generally most effective—approach to improving quality is to set `stringency` to 1.

You can improve quality by increasing `iterations.max` and `fail.iter`.

If the `converged` component of the result is `FALSE`, then it is probably the case that the best solution has not been found. You can restart the optimization from where it left off:

```
> opt1 <- trade.optimizer(prices, varian, ...)
> opt2 <- trade.optimizer(prices, varian,
+    start.sol=opt1, ...)
```

For more thorough optimization, increase `iterations.max` and `fail.iter`:

```
> opt3 <- trade.optimizer(prices, varian, start.sol=opt1,
+    iterations=100, fail=10, ...)
```

Another way of doing this same thing would be:

```
> opt3 <- update(opt1, start.sol=opt1,iterations=100,
+    fail=10)
```

S code note

If an object is a list and has a component named `call` (that is an image of the command that created the object), then the `update` function will recompute the call and change any arguments that are given in the call to `update`. The `update` function can be used on the results of `trade.optimizer` and `random.portfolio`.

A natural thing to do if you want a quality solution is to increase the maximum number of iterations to something like 100 or 1000. That is not the best approach. For very large problems 100 iterations might be useful, but increasing the number of runs and increasing the number of iterations slightly is usually a better approach.

11.5 Testing Optimization Quality

The optimization algorithm is random (pseudo-random to be technical). You get a consistent answer for any particular problem because the same random seed is used by default. You can get different paths (and hence probably different answers) by setting the seed argument to NULL. One way of getting a new optimization is:

```
new.optobj <- update(orig.optobj, seed=NULL)
```

You can create a list of a number of solutions for comparison with commands similar to:

```
> opt.list <- vector("list", 10)
> for(i in 1:10) opt.list[[i]] <- update(orig.opt, seed=NULL)
> summary(unlist(lapply(opt.list, function(x)
+     x$results["objective"])), digits=7)
> summary(unlist(lapply(opt.list, function(x)
+      trade.distance(x, orig.opt))), digits=7)
```

Here we've created an empty list, and filled it up with solutions to the problem. Finally we looked at the distribution of the achieved objective, and the distribution of trade distances from the original solution.

S code note

Some care needs to be used when doing `seed=NULL` in a call to `update`. If `seed` was used in the original call (to `trade.optimizer`), then the result will be to use the default value of `seed` (which is fixed) rather than creating a new random seed.

The `update` function creates its own call to the function and then evaluates that call. Saying `arg=NULL` removes `arg` from the call if it was in the original call.

Chapter 12

Utility

Utility is ignored in random portfolios (but see Section 2.7), however it is central to optimization.

The descriptions of the utilities use a few entities:

- The weight vector w.
- The vector of expected returns α.
- The variance matrix V.
- The transaction costs *C(w)*. This includes the multiplication of the computed costs by `ucost`.

In long-only portfolios all of the elements of w are non-negative and they sum to 1. In long-short portfolios the weights may be negative, and it is the absolute values of the weights that sum to 1—that is, the weight of an asset is its value in the portfolio divided by the gross value of the portfolio.

$C(w)$ is actually an abuse of notation for the transaction cost. Transaction cost obviously depends on the existing portfolio. Cost is not strictly a function of weight unless the gross value for the portfolio is constant.

12.1 Maximum Information Ratio

This utility (which is the default if there is both a variance and expected returns) is specified with the argument:

```
utility = "information ratio"
```

or

```
utility = "exocost information ratio"
```

The generic information ratio problem is to maximize:

$$\frac{\alpha^T w}{\sqrt{w^T V w}}$$

over all vectors w that satisfy the constraints.

This inherently assumes zero costs. There are two approaches (at least) to incorporating costs. The first is to subtract the trading cost from the expected value:

$$\frac{\alpha^T w - C(w)}{\sqrt{w^T V w}}$$

This is what you get with:

```
utility = "information ratio"
```

The second approach is to have a separate term for costs:

$$\frac{\alpha^T w}{\sqrt{w^T V w}} - C(w)$$

You get this second version with:

```
utility = "exocost information ratio"
```

Example

We look at an example of a utility computation:

```
> opt.utilexamp
...
$results
    objective      negutil          cost      penalty
 -0.032086057 -0.032086057  0.004603756  0.000000000
$objective.utility
[1] "information ratio, ucost = 1279"
$alpha.values
       A0
 6.281538
$var.values
       V0
 150.2766
...
```

Now we compute the utility by hand as:

```
> (6.281538 - 1279 * 0.004603756) / sqrt(150.2766)
[1] 0.03208602
```

The sign is different from `negutil` because we are computing the utility and not the negative utility. The last digit is different because we computed with rounded values—that is the sort of numerical error that we would expect. The key thing is to notice that the C function in the formula for the utility is equal to the `cost` element of the `results` component times the value of `ucost`.

12.2 Mean-Variance Utility

Mean-variance utility is invoked with:

```
utility = "mean-variance"
```

With mean-variance utility we maximize:

$$\alpha^T w - \gamma w^T V w - C(w)$$

where γ is the risk aversion parameter and w satisfies all of the constraints.

Many implementations have a one-half in the variance term, meaning that the risk aversion parameter in those cases is a factor of two different than here. Also in some optimizers the parameter used divides the variance rather than multiplies it—in which case it is a risk tolerance parameter.

The mean-variance risk aversion parameter is invariant to annualization. As long as both `expected.return` and `variance` are for the same time scale, it doesn't matter what time scale it is. However risk aversion is affected when `expected.return` and `variance` are for returns that are put into percent. When percent returns are used, the risk aversion is divided by 100.

[Kallberg and Ziemba, 1983] classify risk aversion greater than 3 as very risk averse, 1 to 2 as moderate risk aversion, and less than 1 as risky—these numbers would be divided by 100 for data representing percent returns. Some additional sense of suitable values for the risk aversion parameter may be found in [Burns, 2003b].

All of the above are focused on long-only portfolios. In long-short portfolios the variance is a smaller quantity so it takes a larger risk aversion to have the same effect. In general risk aversion is likely to be larger in long-short situations than for long-only.

12.3 Mean-Volatility Utility

Mean-volatility utility is invoked with:

```
utility = "mean-volatility"
```

With mean-volatility utility we maximize:

$$\alpha^T w - \zeta\sqrt{w^T V w} - C(w)$$

where ζ is the risk aversion parameter and w satisfies all of the constraints.

The mean-volatility risk aversion does depend on annualization. For example the risk aversion for annual data is $\sqrt{252}$ times the risk aversion for daily data. But volatility aversion is invariant to whether or not the data are in percent.

12.4 Minimum Variance

Minimum variance utility is specified with:

```
utility = "minimum variance"
```

The minimum variance utility minimizes:

$$w^T V w + C(w)$$

over the w that satisfy the constraints.

This utility is used when:

- The `utility` argument is set to be `"minimum variance"`.
- The `expected.return` argument is not given (or is `NULL`) and distance is not being minimized.
- The utility is either mean-variance or mean-volatility and `risk.aversion` is `Inf`.

12.5 Maximum Expected Return

The maximum expected return utility is specified via:

```
utility = "maximum return"
```

With this utility we maximize:

$$\alpha^T w - C(w)$$

where w must satisfy the constraints.

This utility is used when:

- The `utility` argument is set to be `"maximum return"`.
- The `variance` argument is not given (or is `NULL`) and distance is not being minimized.
- The utility is mean-variance or mean-volatility and `risk.aversion` is 0.

In the last of these items, the stated utility in the output remains mean-variance or mean-volatility, but a look at the definitions shows that it is really maximum expected return.

12.6 Minimum Distance

This is specified with:

```
utility = "distance"
```

This minimizes the distance from the portfolio to the specified target portfolio plus the trading costs. More details are found in Section 5.4.

12.7 Going Farther

- Chapter 13 discusses utilities where multiple variances and/or expected return vectors are given.

Chapter 13

Advanced Features

This chapter will be of use if you have or want:

- Multiple variances
- Multiple expected return vectors
- Multiple benchmarks
- Compact variances (seldom recommended)

If you merely want constraints on multiple benchmarks, that is easy and is discussed on page 57.

13.1 Multiplicity

The computation that is performed is really controlled by three entities: the alpha table, the variance table and the utility table. In the standard case of no multiplicity, these entities can safely be ignored. When you do have multiplicity, then understanding what these objects are saying is a good idea. If you understand them, then you can check to make sure that it is doing what you want done. If it isn't doing what you want automatically, then you can tell it to do what you do want.

If you are generating random portfolios but not optimizing, then you can skip parts of this chapter. The recommended route in this case is:

- Section 13.2 Alpha and Variance Tables
- Section 13.3 Variance Constraints
- Section 13.4 Expected Return Constraints

Table 13.1 describes arguments to `trade.optimizer` (and `random.portfolio`) that are useful when dealing with multiplicity. All of these arguments have defaults—you only need to give values for these if the default behavior is not what you want.

Table 13.1: Arguments for multiple variances and expected returns.

Argument	Description
`vtable`	gives the variance-benchmark combinations
`atable`	gives the expected return-benchmark combinations
`utable`	describes all the combinations to get utilities
`quantile`	states which quantile of the multiple objectives to use
`dest.wt`	gives weights to the multiple objectives if desired

13.2 Alpha and Variance Tables

The purpose of the alpha table is to state what combinations of expected return vectors and benchmarks will be used in the problem. Likewise the variance table identifies variance-benchmark combinations that will be used.

An alpha table is a matrix with two rows and an arbitrary number of columns. In simple cases the alpha table will look like:

```
> optobj$atable
           A0
alpha       0
benchmark  -1
attr(,"benchmarks")
[1] ""
```

Or if a benchmark is given, it will look something like:

```
> optobj$atable
         A0 -- spx
alpha            0
benchmark      500
attr(,"benchmarks")
[1] "spx"
```

The first row gives the zero-based index of the expected return. If there is only one set of expected returns, then this is always zero. If there are no expected returns, this is negative one.

The second row gives the zero-based index of the corresponding benchmark. A negative number in this row means there is no benchmark.

There is a `benchmarks` attribute which is a character representation of the benchmarks. When you are passing an object in as the `atable` argument, the second row of the matrix itself is overwritten with the information in the `benchmarks` attribute (which must be present).

Variance tables are essentially the same as alpha tables. The first row, of course, corresponds to variances rather than alphas. The other change is that there is a third row which states if this combination is only used in the utility (value 1) or if it is used in constraints as well (value 0). This information is used to more efficiently generate random portfolios. As with alpha tables when you pass an object in as the `vtable` argument, the second row is overwritten with the information given in the `benchmarks` attribute of the object.

Here is an example of the alpha table and variance table from an optimization that has two variance matrices, three expected return vectors, and no benchmarks:

```
> opt1$atable
            A0   A1   A2
alpha        0    1    2
benchmark   -1   -1   -1
attr(,"benchmarks")
[1] "" "" ""
> opt1$vtable
               V0   V1
variance        0    1
benchmark      -1   -1
utility.only    1    1
attr(,"benchmarks")
[1] "" ""
```

Here are the tables from an optimization with the same variance, one expected return but with two benchmarks:

```
> opt2$atable
          A0 -- Ind1 A0 -- spx
alpha              0         0
benchmark          8         9
attr(,"benchmarks")
[1] "Ind1" "spx"
> opt2$vtable
             V0 -- Ind1 V1 -- Ind1 V0 -- spx V0 -- spx
variance              0          1         0         1
benchmark             8          8         9         9
utility.only          1          1         1         1
attr(,"benchmarks")
[1] "Ind1" "Ind1" "spx" "spx"
```

note

The values in the alpha table, variance table and utility table are zero-based because these entities are sucked directly into C where indexing is zero-based rather than one-based as in the S language.

13.3 Variance Constraints

This section discusses the argument:

- `var.constraint`

Simple use of this argument is discussed on page 127. This argument also handles problems that are too complex for the `bench.constraint` argument.

The `var.constraint` argument can be:

- a plain vector (with one or more values) with or without names
- a one–column matrix with or without row names
- a two-column matrix with or without row names

Plain vectors are equivalent to one-column matrices. One-column matrices are equivalent to two-column matrices where the first column is all zeros.

The names on vectors or row names of matrices specify the zero-based index of the column of the variance table. When there are no names, then the first column(s) are implied (in order).

So all six of the following are specifying the same constraints:

```
var.constraint = vc.vec
var.constraint = vc.1mat
var.constraint = vc.2mat
var.constraint = vc.vec.nam
var.constraint = vc.1mat.nam
var.constraint = vc.2mat.nam
```

where:

```
> vc.vec
[1] 2.4 1.5
> vc.1mat
     [,1]
[1,]  2.4
[2,]  1.5
> vc.2mat
     [,1] [,2]
[1,]    0  2.4
[2,]    0  1.5
> vc.vec.nam
  0   1
2.4 1.5
> vc.1mat.nam
  [,1]
0  2.4
1  1.5
> vc.2mat.nam
  [,1] [,2]
0    0  2.4
1    0  1.5
```

All of these are saying the variance-benchmark combination in the first column of the variance table has an upper bound of 2.4 and the variance-benchmark combination in the second `vtable` column is to be no more than 1.5.

If you were to give the equivalent of `vc.vec.nam` directly, you would do:

```
var.constraint = c('0'=2.4, '1'=1.5)
```

A very similar command means something quite different:

```
var.constraint = cbind(2.4, 1.5)
```

means that the (first) variance is constrained to be at most 1.5 but no less than 2.4. (In this case you will get an error, but you can not depend on getting an error when you confuse the direction of matrices.) Note that if the numbers were reversed, the command above would be sensible—maybe not what you want, but sensible.

Two-column matrices give lower bounds as well as upper bounds on the variance. Lower bounds are most likely to be useful in generating random portfolios, but are sometimes of interest in optimization as well.

Consider the matrix:

```
> varcon.m1 <- rbind(c(1.2, 1.4), c(1.3, 1.5))
> varcon.m1
     [,1] [,2]
[1,] 1.2  1.4
[2,] 1.3  1.5
```

Suppose that there are three columns in `vtable` (perhaps the `variance` argument is a three-dimensional array with 3 slices in the third dimension). Then the command:

```
var.constraint = varcon.m1
```

would say that the first variance-benchmark combination is restricted to be between 1.2 and 1.4, the second variance-benchmark is restricted to be between 1.3 and 1.5, and the third is unrestricted.

Now let's add some row names.

```
> varcon.m2 <- varcon.m1
> dimnames(varcon.m2) <- list(c(2, 0), NULL)
> varcon.m2
  [,1] [,2]
2  1.2  1.4
0  1.3  1.5
```

The command:

```
var.constraint = varcon.m2
```

says that the third variance-benchmark is to be between 1.2 and 1.4, the first variance-benchmark is restricted to be between 1.3 and 1.5, and the second variance-benchmark is unconstrained.

Some examples with multiple variances can be found on page 135.

13.4 Expected Return Constraints

This section discusses the argument:

- `alpha.constraint`

Simple use of this argument is discussed on page 55.

Use of this argument is analogous to the advanced use of `var.constraint` (page 127) except that a one-column matrix is a lower bound rather than an upper bound, and indices refer to expected return-benchmark combinations (columns of `atable`) rather than to variance-benchmark combinations.

So for instance:

```
alpha.constraint = c("1"=5.6, "0"=2.3)
```

is equivalent to

```
alpha.constraint = ac.2mat.nam
```

or

```
alpha.constraint = ac.2mat
```

where:

```
> ac.2mat.nam
  [,1] [,2]
1  5.6  Inf
0  2.3  Inf
> ac.2mat
     [,1] [,2]
[1,]  2.3  Inf
[2,]  5.6  Inf
```

13.5 Multiple Utilities

This section does not pertain to generating random portfolios (unless a utility constraint is imposed).

When there are multiple columns in the alpha or variance tables, there are usually multiple utilities for any given trade. In order to do an optimization, we need to combine that set of numbers into just one number.

A common approach is to get a min-max solution. This process consists of always picking out the worst answer for each trade—we then try to get the best of these worst values.

Another approach would be to take the median of the values for a trade. This represents an optimistic attitude, while min-max is very pessimistic—min-max assumes that nature will do its worst against us. The `quantile` argument allows us to choose either of these, or something in between. The useful range of values for `quantile` is 0.5 (the median) to 1 (the maximum—yielding the min-max solution).

Each of the individual utilities for a trade is called a "destination". What goes into a destination is under user control. For example, if some destinations are deemed to be more important than others, they can be given more weight with the `dest.wt` argument.

There are two sorts of destination weight. Each utility has a weight within its destination. The within destination weights can be used to create a weighted average of utilities within a single destination; they can also be used to modify the utility as demonstrated in the dual benchmark example which starts on page 132.

The weights in the `dest.wt` argument, on the other hand, are weights among the destinations which are used along with the quantile argument to control the selection of the objective from among the destinations. For example in scenario analysis (see page 138) `dest.wt` could be the probability assigned to each scenario.

note

The answer when `dest.wt` is not given is, in general, slightly different than when `dest.wt` is given and all of the weights are equal.

The actual problem that is done is mostly specified by the utility table.

13.6 Utility Tables

The utility table is a matrix that gives the combinations of variances, expected returns and benchmarks that are to be used and what to do with them. Here is the utility table from our example of two variances and three expected returns:

```
> opt1$utable
                  [,1] [,2] [,3] [,4] [,5] [,6]
alpha.spot           0    1    2    0    1    2
variance.spot        0    0    0    1    1    1
destination          0    1    2    3    4    5
opt.utility          1    1    1    1    1    1
risk.aversion        1    1    1    1    1    1
wt.in.destination    1    1    1    1    1    1
```

The first row is the expected return—this is the zero-based index of the columns of `atable`. That is, it indicates a particular expected return-benchmark combination. (But note that this is the zero-based location of the distance for columns corresponding to distance utilities.)

The second row is similar to the first except that it is columns of the `vtable`.

The third row is the zero-based index of the destination for the utility with this combination of alpha and variance. The utility for the combination is computed, multiplied by its weight in the destination (the sixth row of the utility table) and added to whatever is there. That is, a destination may hold a weighted average of utilities. If a destination is a negative number, the utility is not placed into a destination—these are combinations that are presumably to be used in a constraint. There must be at least one zero in the destination row, and the destinations must be numbered from zero upwards with no integers skipped.

The fourth row is an indicator for the type of utility to be performed. The allowable codes are given in Table 13.2.

Table 13.2: Utilities codes for the utility table.

Code	Meaning
0	mean-variance
1	information ratio
2	exocost information ratio
3	minimum variance
4	maximum return
5	mean-volatility
6	distance

The fifth row holds the risk aversion parameter for the cases where it is used—the value is ignored in the other cases. When the `utable` argument is given, then the risk aversion in the utility table is used rather than using the value of the `risk.aversion` argument unless `force.risk.aver` is set to `TRUE`.

The example utility table above specifies that six utilities are to be computed and each put into a different destination. Each combination of expected return and variance is used, and the information ratio is the utility in all cases.

13.7 Multiplicity Examples

This section provides a few examples of using multiple variances, expected returns and benchmarks.

Dual Benchmarks

Dual benchmarks can be used, for example, to incorporate predictions about a future rebalancing of the benchmark. Due to the relative movements that have occurred since the last rebalance, we may have predictions of what assets will enter and exit the benchmark and with what weight. The portfolio can be optimized against both benchmarks—this allows the portfolio to be rebalanced to some extent against the new benchmark while still having protection relative to the current one. This can mean a cheaper rebalance if it is done before others do their rebalancing.

If our benchmark is `spx` and our prediction of the post-rebalance benchmark is `newspx`, then minimizing variance relative to the two benchmarks can be done as:

```
> opts1 <- trade.optimizer(prices, varian,
+    long.only=TRUE, gross.value=1e6, existing=cur.port,
+    utility = "minimum variance",
+    benchmark=c("spx", "newspx"), quantile=1)
```

Often a min-max solution is found for dual benchmarks. To get such a solution, set `quantile` to 1. The min-max solution says that we want to minimize the worst tracking error. It is typical (though not always true) that the utility in the optimal solution is the same for both benchmarks:

```
> sqrt(252 * opts1$var.values)
      V0 -- spx  V0 -- newspx
[1] 0.009029673   0.009029442
> opts1$utility.values
[1] 3.235515e-07 3.235350e-07
```

In this example the tracking errors are the same at 90 basis points, and the utilities are equal as well.

A reasonable complaint about this optimization is that it treats both benchmarks equally even though the new benchmark is speculative while the current one is known precisely. It is probably more reasonable to insist on a smaller tracking error against the current benchmark.

A difference in the tracking errors is easily achieved with the addition of another argument. The utility table needs to be changed slightly from its default. So we recover the utility table from the original optimization, make the change, then do a new optimization:

```
> utab1 <- opts1$utable
> utab1
                  [,1] [,2]
alpha.spot           0    0
variance.spot        0    1
destination          0    1
opt.utility          0    0
risk.aversion        1    1
wt.in.destination    1    1
> utab1[6,1] <- 1.4
> utab1
                  [,1] [,2]
alpha.spot         0.0    0
variance.spot      0.0    1
destination        0.0    1
opt.utility        0.0    0
risk.aversion      1.0    1
wt.in.destination  1.4    1
```

All that we have done is change the weight-in-destination value for the first (current) benchmark from 1 to 1.4. Now the optimization yields:

```
> opts2 <- trade.optimizer(prices, varian,
+    long.only=TRUE, gross.value=1e6, existing=cur.port,
+    utility = "minimum variance",
+    benchmark=c("spx", "newspx"), quantile=1,
+    utable=utab1)
> sqrt(252 * opts2$var.val)
      V0 -- spx  V0 -- newspx
[1] 0.008227913   0.009735447
> opts2$utility.val
[1] 3.761031e-07 3.761069e-07
```

The utility values are still equal, but now the utility represents something different for the current benchmark than for the new benchmark, so the tracking errors are different—now they are 82 basis points for the current benchmark and 97 for the new benchmark.

There is not a limit on the number of benchmarks that you can use—just add the names to the vector given as the `benchmark` argument.

If you have an active portfolio, you may want to put on constraints relative to each benchmark. This is done precisely the same as with a single benchmark constraint. Adding a second benchmark constraint is just like adding any other constraint—it does not create multiple utilities. An example is given on page 75.

Benchmark-relative Utility and Absolute Variance Constraint

We saw (on page 74) that it is easy to have a utility in absolute terms on the portfolio with a benchmark constraint. The reverse problem—benchmark-relative utility with a constraint on portfolio volatility—is just as easy to state, but not as easy to do.

We need to create a variance table to pass in as the `vtable` argument. It needs a column with the benchmark to use in the utility and a second column without the benchmark for the constraint. We can do this with:

```
> vtab.buav <- rbind(c(0,0), c(0,-1), c(1,0))
> attr(vtab.buav, "benchmarks") <- c("spx", "")
```

We will also need to create a utility table, but in this case it is simple enough to create on the fly within the call to the optimizer. Our optimizer call can look something like:

```
> opt.buav <- trade.optimizer(prices, varian,
+     expected.return=alphas, benchmark='spx',
+     var.constraint=c('1'=100), vtable=vtab.buav,
+     utable=rep(c(0,1), c(3,3)), ... )
```

We can check the tables that are created to see if it is doing what we want.

```
> opt.buav$atable
          A0 -- spx
alpha             0
benchmark       500
attr(,"benchmarks")
[1] "spx"
> opt.buav$vtable
             V0 -- spx      V0
variance             0       0
benchmark          500      -1
utility.only         1       0
attr(,"benchmarks")
[1] "spx" ""
> opt.buav$utable
                    [,1]
alpha.spot             0
variance.spot          0
```

```
destination         0
opt.utility         1
risk.aversion       1
wt.in.destination   1
```

The utility table we created says to maximize the information ratio. We need to put a name on the object given as the variance constraint because we want to constrain the variance-benchmark combination given in the second column of the variance table (not the first column).

Rival Variance Forecasts

If you have two or more variance matrices, you can use these in a single optimization. For example if you have a statistical factor model, a fundamental factor model and a macroeconomic factor model, you could use all three in the optimization. Since these are created using different sources of information, it is reasonable to suppose that results may be better by using more than one of them.

The first task is to create a suitable variance object, which will (almost always) be a three-dimensional array. We can use the threeDarr function from the BurStFin package to do this:

```
> varmult <- threeDarr(vstat, vfund, vmacro,
+    slicenames=c('stat', 'fundamental', 'macro'))
```

S code note

A three-dimensional array is a generalization of a matrix—instead of a dim attribute that has length 2, it has a length 3 dim. Likewise, the dimnames is a list that has three components.

We need each of the variance matrices to be in a slice of the third dimension.

You need to decide which quantile to optimize. The best quantile to use probably depends on how the variance matrices relate to each other and on the type of optimization that you are performing. Once this decision is made, the optimization is performed as with a single variance:

```
> opts.m1 <- trade.optimizer(prices, varmult,
+    long.only=TRUE, gross.value=1e6, expected.return=alphas,
+    bench.constrain = c(spx=.05^2/252), quantile=.7)
```

This command is maximizing the information ratio with a tracking error constraint as is done on page 74. In this case the tracking error is constrained relative to each of the variance matrices. The information ratio that is being optimized is the 70% quantile of the set of information ratios using each of the variances.

If you are doing a variance constraint, for instance maximizing the information ratio in a long-short portfolio as on page 83, then the approach is slightly different. You need to make the variance constraints argument the proper length yourself:

```
> opts.m2 <- trade.optimizer(prices, varian,
+    expected.return=alphas, gross.value=1e6,
+    net.val=c(-1e4, 2e4), ntrade=55, max.weight=.05,
+    var.constraint=rep(.04^2/252, 3))
```

If the value given as the variance constraint weren't replicated, then only the first variance would be constrained.

The `var.constraint` argument is the more general. Benchmark constraints actually create variance constraints—the `bench.constraint` argument only exists for convenience. (When more than one variance is given, `bench.constraint` puts the constraint on each of the variances.) More control can be gained over variance constraints by putting names on the vector given. The names need to be the zero-based index of the columns of the variance table (see page 127). For example to constrain the first variance to a volatility of 4% and the third to a volatility of 5%, the command would be:

```
> opts.m3 <- trade.optimizer(prices, varian,
+    expected.ret=alphas, gross.val=1e6,
+    net.val=c(-1e4, 2e4), ntrade=55, max.weight=.05,
+    var.constraint=c("0"=.04^2/252, "2"=.05^2/252))
```

S code note

The names given inside the `c` function need to be put in quotes because they are not proper names of S objects. S would try to interpret them as numbers if they were not in quotes.

If any element of the `var.constraint` argument is named, then they all need to be. When there are no names, the columns of the variance table are used in order.

A standard use of multiple variances is to optimize with one, and check the tracking error or volatility with the other. One good reason to evaluate the solution on a variance not used in the optimization is because of the bias created in the optimization process—the tracking error or volatility that is achieved in the optimization is a downwardly biased estimate of what will be realized. Using multiple variances in the optimization will reduce the amount of bias in any particular variance matrix, though some bias will still remain.

Multiple Time Periods

You may have forecasts for multiple time periods. For example, you may have GARCH forecasts of the variance matrix for several different time horizons. You want the portfolio to do well at all of the time scales. A key issue is how to weight the different time scales. You may want different risk profiles at different time scales—for instance, expected returns may be given relatively more weight versus risk for short time horizons.

Credit Risk

The variance matrices that we have talked about have all been measures of market risk. There are other sources of risk—credit risk, for example—that could be included in an optimization. For more on credit risk, see for example [Gupton et al., 1997]. While we'll speak of credit risk here, keep in mind that other forms of risk could be used instead or in addition.

Suppose that we have `var.market` and `var.credit` that represent the two types of risk. While it may be possible to scale the credit risk so that it is comparable to the market risk, it may be easier to accept that they are different. If the two forms of risk are not comparable, then you can perform the optimization with the market risk as you normally would and put a constraint on the credit risk.

When deciding how to constrain the credit risk, the first thing to do is explore its limits. One end of the range is when credit risk is ignored entirely in the optimization, the other end is when the credit risk is minimized with no concern for the market risk.

```
> op.cred0 <- trade.optimizer(prices, var.market, ...)
> trade.optimizer(prices, var.credit, ...,
+     funeval=0, start.sol=op.cred0)$var.value
[1] 26.02844
> trade.optimizer(prices, var.credit,
+     utility="minimum variance", ...)$var.value
[1] 11.92467
```

So the range of interest is about 12 to 26. If we decided to constrain the credit risk to 18, then we would do:

```
> var.mult <- threeDarr(var.market,var.credit,
+      slicenames=c('market', 'credit'))
> op.cred1 <- trade.optimizer(prices, var.mult, ...,
+      var.constraint=c("1"=18))
```

The first command creates the three dimensional array containing the two types of risk. The second command does the optimization. The variance constraint is a named vector where the name is the zero-based index to the third dimension of the variance array. In this case the name is "1" because we are constraining the second variance.

But `op.cred1` is not the optimization that we are aiming for. The credit risk is used in the utility as well as being constrained. We need to tell the optimizer not to use credit risk in the utility by passing in a `utable` argument. We can make a simple change to the utility table from the optimization we've just done.

```
> op.cred1$utable
                [,1] [,2]
alpha.spot        0    0
variance.spot     0    1
destination       0    1
opt.objective     1    1
risk.aversion     1    1
wt.in.destination 1    1
> uticred <- op.cred1$utable
> uticred[3,2] <- -1
> uticred
```

```
[,1] [,2]
alpha.spot        0    0
variance.spot     0    1
destination       0   -1
opt.objective     1    1
risk.aversion     1    1
wt.in.destination 1    1
>
> op.cred2 <- trade.optimizer(prices, var.mult, ...,
+      var.constraint=c("1"=18), utable=uticred)
```

The change we made was to say that the column with credit risk in it shouldn't go into a destination. One indication that we are doing it right is that the `utility.values` component of `op.cred2` has length one, while it is length two for `op.cred1`. Since we only want one variance in the utility, the utility that we want will have length one.

Multiple Scenarios

Scenario analysis creates some hypotheses about what might happen in the future, and then tries to pick the best course of action. The assumption is that the future will look like one of the scenarios, or perhaps some combination of scenarios.

Our first step is to create some data. We will create a "medium" scenario, a "crash" scenario and a "prosperous" scenario. (The probability that these numbers are reasonable is close to zero.)

```
> anam4 <- c("equities", "bonds", "commodities", "cash")
> acor.medium <- matrix(c(1, .2, .1, .2, 1, .1, .1, .1, 1),
+     3, 3)
> acor.medium
            equities bonds commodities
equities         1.0   0.2         0.1
bonds            0.2   1.0         0.1
commodities      0.1   0.1         1.0
 > acor.crash <- matrix(c(1, -.3, .3, -.3, 1, -.2, .3,
+    -.2, 1), 3, 3)
> acor.crash
     [,1] [,2] [,3]
[1,]  1.0 -0.3  0.3
[2,] -0.3  1.0 -0.2
[3,]  0.3 -0.2  1.0
> acor.prosper <- matrix(c(1, .4, .3, .4, 1, .2, .3,
+    .2, 1), 3, 3)
> acor.prosper
     [,1] [,2] [,3]
[1,]  1.0  0.4  0.3
[2,]  0.4  1.0  0.2
[3,]  0.3  0.2  1.0
```

```
> avol.medium <- c(20, 8, 12)
> avol.crash <- c(35, 16, 20)
> avol.prosper <- c(15, 5, 7)
> avar.medium <- rbind(cbind(t(acor.medium * avol.medium) *
+    avol.medium, 0), 0)
> avar.crash <- rbind(cbind(t(acor.crash * avol.crash) *
+    avol.crash, 0), 0)
```

S code note

The last command creating `avar.crash` does several things at once. First (in the center of the command) it multiplies the correlation matrix times the volatility vector which multiplies each row of the correlation matrix by the corresponding volatility. Then it transposes the matrix and multiplies again by the volatility vector—this is now multiplying what originally were the columns of the correlation matrix by their corresponding volatilities. Then it binds on a column of all zeros (for cash), and finally binds on a row of all zeros. Below you can see the result of all this manipulation once the asset names are put onto the variance matrix.

```
> avar.prosper <- rbind(cbind(t(acor.prosper * avol.prosper)
+    * avol.prosper, 0), 0)
> dimnames(avar.medium) <- list(anam4, anam4)
> dimnames(avar.crash) <- list(anam4, anam4)
> dimnames(avar.prosper) <- list(anam4, anam4)
> avar.medium
            equities bonds commodities cash
equities         400  32.0        24.0    0
bonds             32  64.0         9.6    0
commodities       24   9.6       144.0    0
cash               0   0.0         0.0    0
> avar.crash
            equities bonds commodities cash
equities        1225  -168         210    0
bonds           -168   256         -64    0
commodities      210   -64         400    0
cash               0     0           0    0
> avar.prosper
            equities bonds commodities cash
equities       225.0    30        31.5    0
bonds           30.0    25         7.0    0
commodities     31.5     7        49.0    0
cash             0.0     0         0.0    0
> aret.medium <- c(8, 4, 5, 3)
> aret.crash <- c(-20, 6, -9, 3)
> aret.prosper <- c(25, 3.5, 8, 3)
> names(aret.medium) <- anam4
> names(aret.crash) <- anam4
> names(aret.prosper) <- anam4
> aret.crash
   equities       bonds commodities        cash
        -20           6          -9           3
> aret.prosper
   equities       bonds commodities        cash
       25.0         3.5         8.0         3.0
```

Since we are doing asset allocation, there are no prices for the assets and we want the answer to be the weights of the assets. Hence we want to create a "price" vector that is all ones and set the gross value to a useful value.

```
> aprice <- rep(1, 4)
> names(aprice) <- anam4
> aprice
   equities       bonds commodities        cash
          1           1           1           1
> ten.thou <- 10000 + c(-.5, .5)
> op.medium <- trade.optimizer(aprice, avar.medium,
+    aret.medium, gross.value=ten.thou, long.only=TRUE,
+    utility="mean-variance", risk.aversion=.02)
> op.medium$new.portfolio / 100
    equities       bonds commodities        cash
       27.86       20.83       28.69       22.62
```

Above we have performed a mean-variance optimization with the "medium" data, and then shown the weights of the resulting portfolio in percent. (The variance matrix is not positive definite but the optimizer doesn't care—note, though, that no more than one zero variance should be in a single variance matrix.)

In order to perform simultaneous optimization, we need to put the three variance matrices into a three-dimensional array, and the expected return vectors into a matrix:

```
> avar.all <- threeDarr(avar.crash, avar.medium, avar.prosper,
+    slicenames=c('crash', 'medium', 'prosper'))
```

S code note

A three-dimensional array is a generalization of a matrix—instead of a `dim` attribute that has length 2, it has a length 3 `dim`. Likewise, the `dimnames` is a list that has three components.

We need each of the variance matrices to be in a slice of the third dimension.

```
> aret.all <- cbind(crash=aret.crash, medium=aret.medium,
+    prosper=aret.prosper)
> aret.all
            crash medium prosper
equities      -20      8    25.0
```

```
bonds            6      4     3.5
commodities     -9      5     8.0
cash             3      3     3.0
```

Min-Max Solution

The most common approach to simultaneous optimization is to maximize the minimum utility—also known as Pareto optimality. To clarify: each allocation will produce a utility for each scenario, the optimizer only pays attention to the worst utility for an allocation (with no regard to which scenario produces that utility); it then finds the allocation that does best with respect to the worst utility. The aim, then, is to never do really badly.

Before we do the actual optimization, we need to do some ground work.

```
> op.0 <- trade.optimizer(aprice, avar.all, aret.all,
+    gross.value=ten.thou, long.only=TRUE,
+    utility="mean-variance", iterations=0)
  # (warning message omitted)
> utab3 <- op.0$utable
> utab3
                 [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
alpha.spot          0    1    2    0    1    2    0    1    2
variance.spot       0    0    0    1    1    1    2    2    2
destination         0    1    2    3    4    5    6    7    8
opt.utility         0    0    0    0    0    0    0    0    0
risk.aversion       1    1    1    1    1    1    1    1    1
wt.in.destination   1    1    1    1    1    1    1    1    1
```

The default behavior is to get all of the combinations of expected returns and variances—in this case we have three of each so there are 9 combinations. For our scenario analysis, though, we want only the combinations where the expected returns and the variances match—three combinations. In order to get the behavior that we want, we need to provide a suitable matrix as the `utable` argument. We could build a matrix from scratch, but it is slightly easier to start with the one that is wrong.

There are three changes that we need to make to this matrix. We need to select the three columns that we want, we need to reset the destinations so that they are numbered from zero through one less than the number of combinations, and we should specify the risk aversion parameter that we want.

```
> utab3 <- utab3[, c(1, 5, 9)]
> utab3["destination", ] <- 0:2
> utab3["risk.aversion", ] <- 0.02
> utab3
                  [,1] [,2] [,3]
alpha.spot        0.00 1.00 2.00
variance.spot     0.00 1.00 2.00
destination       0.00 1.00 2.00
opt.utility       0.00 0.00 0.00
risk.aversion     0.02 0.02 0.02
wt.in.destination 1.00 1.00 1.00
```

At this point we are ready to do the actual optimization. In addition to needing the `utable` argument, the `quantile` argument needs to be 1 in order to get a min-max solution:

```
> op.minmax <- trade.optimizer(aprice, avar.all, aret.all,
+    gross.value=ten.thou, long.only=TRUE,
+    utility="mean-variance", utable=utab3, quantile=1)
> op.minmax$new.portfolio / 100
equities    bonds     cash
    0.91    34.82    64.27
```

A look at the utility values shows typical behavior of a min-max solution—the solution has the same utility for two of the scenarios.

General Simultaneous Optimization

Min-max optimization protects against the very worst outcome. However, it may really be buying too much insurance. We can change the `quantile` argument to a number that is less than 1 (but only numbers that are at least one-half are sensible):

```
> op.q7 <- trade.optimizer(aprice, avar.all, aret.all,
+    gross.value=ten.thou, long.only=TRUE,
+    utility="mean-variance", utable=utab3,
+    quantile=.7)
> op.q7$new.portfolio / 100
equities    bonds     cash
    0.76    32.70    66.54
```

In this case (of only three utility values) the quantity it is minimizing is a weighted average of the two worst utilities. For this example there is very little change from the min-max solution (and it seems to go more conservative rather than less conservative).

We can adjust the risk aversion but when `utable` is given, we need to use the `force.risk.aver` control argument. By default the risk aversion in the utility table that is passed in is taken as what is wanted.

```
> op.q7r1 <- trade.optimizer(aprice, avar.all, aret.all,
+    gross.value=ten.thou, long.only=TRUE,
+    utility="mean-variance", utable=utab3, quantile=.7,
+    risk.aversion=.01, force.risk.aver=TRUE)
> op.q7r1$new.portfolio / 100
equities    bonds     cash
    1.52    65.41    33.07
```

13.8 Compact Variance Objects

Usually the full variance matrix is given. In the case of multiple variances, a three-dimensional array is given where the third dimension represents the different variance matrices. Computationally, full matrices are the fastest, so the preferred format is full matrices as long as the memory of the machine on which the optimization is being done is large enough. This will almost certainly be the case unless the universe of assets is very large, or there is a large number of variance matrices.

caution

Unlike when a full matrix is given, there is no mechanism when giving compact variances for the optimizer to ensure that the assets are in the correct order within the variance. The user needs to take full responsibility that the asset order in the variances is the same as in the `prices` argument. This includes benchmarks, which enjoy some automatic treatment when full matrices are given.

In addition to full matrices, there are three formats supported:

- *simple factor models.* These have formula $\Lambda\Lambda^T + \Psi$ where Λ is a matrix of loadings with dimensions that are the number of assets by the number of factors, and Ψ is a diagonal matrix containing the specific variances. This will most likely be the result of a statistical factor model.
- *full factor models.* These have formula $\Lambda\Phi\Lambda^T + \Psi$ where Λ is a matrix of loadings with dimensions that are the number of assets by the number of factors, Φ is the variance matrix of the factors among themselves, and Ψ is a diagonal matrix containing the specific variances.
- *vech.* This is the lower triangle of the variance matrix stacked by column. So there is the variance of the first asset and the covariances of the first asset with all of the other assets, then the variance of the second asset and the covariances of the second asset with assets 3 and onward, etc.

The Variance List

When at least one variance is not full, the `variance` argument to the optimizers needs to be a list with special components. The components of the list are:

- `vardata`: Always required. This is a vector of the actual numbers for the variance representations all concatenated together. The order of the data in the representations is:
  - *full matrix*: The first column, the second column, etc. (This is, of course, the same as the first row, the second row, etc.)
  - *simple factor model*: The loadings for the first asset, the loadings for the second asset, etc. Then the specific variances.
  - *full factor model*: Each column of the factor variance matrix in turn. Then the loadings for the first asset, the loadings for the second asset, etc. Then the specific variances.
  - *vech*: The vech representation.

- `vartype`: Always required. This is a vector of integers with length equal to the number of variance matrices represented. The integers indicate the type of representation for each matrix—there is no restriction on the combinations of types that may be used. The codes for the formats are:
  - *full matrix*: 0
  - *simple factor model*: 1
  - *full factor model*: 2
  - *vech*: 3
- `nvarfactors`: Required if any representation is a factor model (either simple or full). When given, this must have length equal to the length of `vartype`. The elements of this vector that correspond to a factor model must contain the number of factors for that model.
- `varoffset`: Never required, but may add some safety. This is an integer vector containing the offset for the start of each variance matrix within the `vardata` component. The first element is zero, the second is the length of the data for the first variance matrix, the third is the combined lengths of the first and second variance matrices, etc. The length of this vector is the number of matrices given, so the element that would be the total length of `vardata` is not given.

Here is an example of putting an object produced by `factor.model.stat`(from the `BurStFin` package) into this format:

```
> varfac <- factor.model.stat(retmat, out="fact")
> vdata <- c(t(varfac$sdev * varfac$loadings),
+     varfac$uniqueness * varfac$sdev^2)
> vlist <- list(vardata=vdata, vartype=1,
+     nvarfactors=ncol(varfac$loadings))
```

The `vdata` object is a vector of the pertinent numbers in the correct order. Note that the loadings are first multiplied by the standard deviations, then this matrix is transposed. Likewise the uniquenesses need to be scaled by the squared standard deviations. Since this is producing a simple factor model, the type of variance is 1. Finally, the number of factors is given as the number of columns of the loadings matrix. The `vlist` object is ready to be passed in as the `variance` argument.

caution

Be careful when annualizing a factor model representation. You want each element of the actual variance to be multiplied by some number—such as 252—but not every number in the representation is multiplied by the same thing.

Chapter 14

Dregs

This chapter contains topics that don't seem to belong anywhere else.

14.1 Portfolio Probe Constituents

The `pprobe.verify` function is useful for seeing if the installation on a particular machine worked okay. It is far from a full test suite—it merely checks that all of the functions are present, and that one particular problem optimizes okay. This can also be used to see which version of Portfolio Probe is present.

14.2 The Objectives

The objective function of an optimization has a numeric value as its result, and a set of inputs that can be changed. The job of the optimizer is to find the combination of inputs that achieves the best result for the objective function. By convention optimizers find the minimum of objective functions. The optimizer in `trade.optimizer` follows the convention.

The objective in `trade.optimizer` is the sum of two quantities: the negative of the utility (except "minimum variance" and "minimum distance" don't take the negative), and the penalty for broken constraints.. The penalty for a broken constraint is the appropriate element of `penalty.constraint` times a measure of how broken the constraint is. The penalties for all of the constraints are summed to get *the* penalty.

The same process is done for generating random portfolios. The difference is that the utility part is zero when `random.portfolio` is used—so there is only the penalty part that is minimized.

14.3 Writing C or C++ Code

The C code that underlies the optimization and random portfolio generation can be incorporated into your own C or C++ programs. There are quite a few arguments to the C functions for optimization, and some of the arguments involve a number of variables. Thus it is far from trivial to write the program calling these functions.

A much more practical approach to using Portfolio Probe within a C++ environment is to use the `RInside` R package to call the R functions in Portfolio Probe. So C++ calls R which calls C and results are available back in the original C++. See the cookbook section of the Portfolio Probe website for a simple example.

14.4 Bug Reporting

Send email to support@portfolioprobe.com to report any bugs that you find. A report should include:

- The operating system (including the version) that you are using.
- The version of R or S-PLUS that you are using.
- Give the `version` component (optimization) or attribute (random portfolios). For example:

```
> op$version
                C.code                   S.code
  "portgen_BurSt 1.24" "trade.optimizer 029.029"
```

```
> attr(randport, "version")
                 C.code                  S.code
  "randport_BurSt 1.24" "random.portfolio 015"
```

- A full description of the problem—what happens, and what you wanted to happen.
- If S creates an error, then give the results of the call:

```
> traceback()
```

  This displays the state that S was in when the error occurred.
- If possible, send the data for a small problem that reproduces the bug.
- If the bug is sporadic and uses `random.portfolio`, then also give the random seed from an object where the bug is observed:

```
> attr(ranport, "seed")
```

A reward is given to the first to report a bug in the code or documentation. Ideas for improving the ease of use or functionality are always welcome.

Bibliography

[Almgren et al., 2005] Almgren, R., Thum, C., Hauptmann, E., and Li, H. (2005). Equity market impact. *Risk*.

[Burns, 1998] Burns, P. (1998). *S Poetry*. http://www.burns-stat.com.

[Burns, 2003a] Burns, P. (2003a). On using statistical factor models in optimizing long-only portfolios. Working paper, Burns Statistics, http://www.burns-stat.com/.

[Burns, 2003b] Burns, P. (2003b). Portfolio sharpening. Working paper, Burns Statistics, http://www.burns-stat.com/.

[Burns, 2011] Burns, P. (2011). The R Inferno. Tutorial, Burns Statistics, http://www.burns-stat.com/.

[Burns et al., 1998] Burns, P., Engle, R., and Mezrich, J. (1998). Correlations and volatilities of asynchronous data. *The Journal of Derivatives*, 5(4).

[diBartolomeo, 2003] diBartolomeo, D. (2003). Portfolio management under taxes. In Satchell, S. and Scowcroft, A., editors, *Advances in Portfolio Construction and Implementation*. Butterworth–Heinemann.

[Grinold and Kahn, 2000] Grinold, R. C. and Kahn, R. N. (2000). *Active Portfolio Management*. McGraw–Hill.

[Gupton et al., 1997] Gupton, G. M., Finger, C. C., and Bhatia, M. (1997). *CreditMetrics ^{TM}—Technical Document*. http://www.riskmetrics.com.

[Kallberg and Ziemba, 1983] Kallberg, J. G. and Ziemba, W. T. (1983). Comparison of alternative utility functions in portfolio selection problems. *Management Science*, 29:1257–1276.

[Michaud, 1998] Michaud, R. O. (1998). *Efficient Asset Management*. Harvard Business School Press.

Index

120/20 portfolio, 30

abbreviation of function arguments, 20
allowance argument, 29, 30
alpha table, 126–127
alpha.constraint argument, 55
alphas, *see* expected returns
amortize cost, 99
annualization
  information ratio, 74
  risk aversion, 123
argument abbreviation, 20
array
  three-dimensional, 135, 140, 142
as.matrix function, 89, 90
asset allocation, 79
assets used in optimization, 71, 81
asynchronous data, 91, 103
atable, 131
atable argument, 126

bench.constraint argument, 56–57, 75, 135
bench.weights argument, 34, 72, 75, 92, 103, 105, 107
benchmark
  add to variance, 92
  alpha table, 126–127
  constraint, 74–75, 102, 135
  dual, 75, 132–134
  variance table, 126–127
benchmark for long-short, 113–114
benchmark-relative utility, 134
bug report, 146
build.constraints function, 43
BurStFin package, 90
buy-hold-sell list, 77, 83

c function, 59, 62, 75
C or C++ code, 145–146

call component, 118
cash flow, 78–79, 85–86
CAUTION, 22, 32, 33, 91, 93, 96, 143, 144
cbind function, 45, 110
cheatsheet, 107–108
class function, 25
close.number argument, 61
comma-separated file, 24, 89, 94
compact variance representation, 142–144
compare to another program, 116
constraint
  absolute linear, 48–49
  all, 27
  benchmark, 74–75, 102, 135
  building linear, 42–48
  close number, 61
  correlation asset to portfolio, 35
  cost, 60
  count, 52–54
  country, 20
  distance, 57–59
  expected return, 55, 129–130
  linear, 42–52
  liquidity, 33, 39–40
  long-side linear, 49
  market capitalization, 47
  monetary, 27–30
  number long or short, 54
  numerical linear, 46
  penalty for breaking, 64, 93, 145
  portfolio size, 73, 82
  portfolio value, 86
  quadratic, 61–64, 100
  risk fraction, 33
  sector, 20
  short-side linear, 49
  soft, 64
  sum of largest weights, 59–60
  threshold, 37–38, 41

tracking error, 20, 56–57, 74–75, 135
trade, 48–49
trade number, 73
turnover, 73, 75, 83
unsatisfiable, 25–26
utility, 26
variance, 55, 83, 127–129, 135
variance partition, 33, 46
violation, 50–51
weight, 31–60
constraints.realized function, 50
continuously compounded return, 91
conventions, typography, 18
convergence, 105
correlation of assets to portfolio, 35
costs, 95
amortization, 99
asymmetric, 96
constraint, 60
linear, 96
nonlinear, 97
piecewise linear, 100
power law, 98
relative to expected returns, 98–99
relative to noise, 118
square root, 98
taxes, 99–100
count constraint, 52–54
country constraint, 20
cov.wt function, 90
credit risk, 136–138
csv file, 24, 89, 94
currency, 101

data frame, 89
debugger function, 111
debugging S, 110–111
decay, time, 100
deport function, 23
deport function export function, 94
dest.wt argument, 130, 131
destination
of utility, 131
weight, 130, 131
dispersion, 35
distance
constraint, 57–59
optimization, 77–78

do.call function, 23
do.warn argument, 104–107
dollar neutral portfolio, 30
drop function, 89, 90
dual benchmarks, 75, 132–134
duration, 46

enforce.max.weight argument, 32
error produced by S, 146
examples, 72–79, 82–86, 138–142
existing argument, 103
expected returns, 74
multiple, 130, 132–142
relative to costs, 98–99

factor model, 135
full, 143
fundamental, 135
macroeconomic, 135
simple, 143, 144
statistical, 90–92, 143
factor vs factor, 91
factor.model.stat function, 90, 144
fail.iter argument, 25, 118
feasible solution, 51
file
read, 89
write, 23–24, 94
fitted function, 92
force.risk.aver argument, 132
forced trade
automatic, 32
positions argument, 40
forced.trade argument, 38, 84
funeval.max argument, 116

garbage, 101
GARCH, 136
gen.fail argument, 25
generating random portfolios, 19–26
genetic algorithm, 117
global data, 91, 103
gross value
definition, 30
gross.value argument, 29

head function, 21

illiquid assets, 103
import data into S, 89–90
information ratio

annualizing, 74
definition, 74
maximize, 74–75, 82, 121
init.fail argument, 25
intersect function, 45
inventory model, 98
iterations.max argument, 25, 116

leverage, 86
library function, 15
limit.cost argument, 60
lin.abs argument, 49
lin.bounds argument, 43
lin.constraint argument, 43
lin.direction argument, 50
lin.style argument, 45, 52
lin.trade argument, 48
linear constraint, 42–52
building, 42–48
liquidity, 32–33, 39–40, 95
log return, 91
long and short lists, 83
long value
definition, 29
long-only portfolio, 71–80
long-only with shorts, 113
long-short portfolio, 81–88
long-side linear constraint, 49
lot
round, 27
size, 94
lower.trade argument, 32, 77, 84

machine memory, 143
marginal contribution to the benchmark, 34
market impact, 95
market risk, 136
max.weight argument, 31, 32, 82, 109
maximize information ratio, 74–75, 82, 121
maximum expected return utility, 124
maximum weight, 32
mean-variance
optimization, 76, 84
mean-variance utility, 123
mean-volatility utility, 123
median optimization, 130
memory of the machine, 143
min-max optimization, 130, 132, 141–142
minimize tracking error, 72
minimize variance, 72
minimum variance utility, 123
miniter argument, 25
missing values
variance estimation, 92
monetary constraints, 27–30, 86
multiple expected returns, 130, 132–142
multiple time periods, 136
multiple variances, 129, 132–142

net value
definition, 30
nonlinear costs, 97
not optimizing starting solution, 115–116
ntrade argument, 36, 73, 77, 84

objective function, 124, 145
operating system, 146
optimization
algorithm, 117
distance, 77–78
mean-variance, 76, 84
median, 130
min-max, 130, 132, 141–142
not doing, 115–116
objective, 124, 145
Pareto, *see* min-max optimization
real-time, 86–88
utility-free, 77–78
wrong, 101–104
out.trade argument, 19, 20

Pareto optimization, *see* min-max optimization
passive portfolio, 72–73, 132–134
penalty for broken constraint, 93, 145
port.size argument, 36, 73, 75, 82
portfolio
long-only, 71–80
long-short, 81–88
passive, 72–73, 132–134
position
too small, 109

positions argument, 38–42, 108
pprobe.verify function, 145
prices argument, 71, 81

quadratic constraint, 61–64, 100
quantile argument, 130, 135, 142
quotes in S, 59, 136

R language
    instance of S language, 14
R website, 14
random portfolio, 19–26, 116
random portfolio summary, 23
random.portfolio function, 19–27
random.portfolio.utility function, 26
randport.eval function, 22–23, 51
rbind function, 110
read a file, 89
read.table function, 89
real-time optimization, 86–88
rep function, 77, 83, 84
repeat (S construct), 88
return
    continuously compounded, 91
    log, 91
    simple, 91
rf.style argument, 34
rights issue, 102
risk
    credit, 136–138
    market, 136
risk aversion
    invariance of mean-variance, 123
    invariance of mean-volatility, 123
    parameter, 123, 132
risk tolerance, 123
risk.aversion argument, 132
risk.fraction argument, 33–35, 73
rival forecasts, 135–136
round lot, 27

S code note, 18–20, 23, 24, 45, 47, 59, 62, 77, 84, 88, 89, 91, 118, 135, 136, 139, 140
S function
    as.matrix, 89, 90
    build.constraints, 43
    c, 59, 62, 75
    cbind, 45, 110
    class, 25
    constraints.realized, 50
    cov.wt, 90
    debugger, 111
    deport, 23, 94
    do.call, 23
    drop, 89, 90
    factor.model.stat, 90, 144
    fitted, 92
    head, 21
    intersect, 45
    library, 15
    pprobe.verify, 145
    random.portfolio, 19–27
    random.portfolio.utility, 26
    randport.eval, 22–23, 51
    rbind, 110
    read.table, 89
    rep, 77, 83, 84
    summary, 23, 93
    tail, 21
    threeDarr, 135, 137, 140
    traceback, 111, 146
    trade.optimizer, 71–88
    update, 118
    valuation, 65–69
S language
    debugging, 110–111
    includes R, 14
    multi-line command, 18
S-PLUS website, 14
scale.cost argument, 99
scenario analysis, 138–142
sector constraint, 20
sense, making, 101
short value
    definition, 29
short-side linear constraint, 49
shrinkage, *see* Bayesian
simple return, 91
small portfolio, 109
soft constraint, 64
split, stock, 102
stale prices, 103
start
    not optimizing, 115–116
start.sol argument, 103, 116
statistical factor model, 90–92, 143, 144
stock split, 102
stringency argument, 115, 118

sum of largest weights constraint, 59–60
summary function, 23, 93

tab-separated file, 89, 94
tail function, 21
taxes, 99–100
three-dimensional array, 135, 140, 142
threeDarr function, 135, 137, 140
threshold argument, 37–38, 107
threshold constraint
  positions argument, 41
time decay, 100
tol.positions argument, 41
traceback function, 111, 146
tracking error, 133
  constraint, 20, 56–57, 74–75
  minimize, 72
trade
  too trivial, 109
trade.optimizer function, 71–88
trading costs, *see* costs
transaction costs, *see* costs
troubleshooting, 109–110
turnover, *see* trade.value argument
turnover argument, 29, 73, 75, 82, 83, 85
txt file, 89, 94
typography conventions, 18

unchanging utility, 109
universe.trade argument, 32, 41, 77, 84, 85
update function, 118
upper.trade argument, 32, 77, 84
utable argument, 131, 142
utility
  benchmark-relative, 134
  constraint, 26
  definitions, 121
  destination, 131
  table, 131–132, 141–142
  unchanging, 109
  zero, 109
utility-free optimization, 77–78

valuation
  function, 65–69
  too small, 109
value constraints, 86
var.constraint argument, 55, 104
variance, 90–92
  adding benchmark, 92
  argument, 143–144
  compact representation, 142–144
  constraint, 55, 83, 127–129, 135
  factor model, *see* factor model
  fraction, 33–35
  minimize, 72
  missing value treatment, 92
  multiple, 129, 132–142
  table, 126–127, 131
  vech representation, 143
variance partition constraint, 46
vech representation, 143
version component, 146
violation of constraints, 50–51
volume, average daily, 33, 98
vtable argument, 126

warnings, suppressing, 104–107
weight
  constraint, 31–60
  destination, 130, 131
  sum of largest constraint, 59–60
write a file, 23–24, 94
wrong optimization, 101–104

zero utility, 109