

POP Portfolio Construction User's Manual

Burns Statistics

1st December 2005

Copyright 2003-2005 Burns Statistics Limited. All rights reserved.
<http://www.burns-stat.com/>
Edition 3.01: 2005 December 01

S-PLUS is a registered trademark of Insightful Corporation.

Contents

1	Introduction	13
1.1	The Role of Portfolio Optimization	13
1.2	Necessary Tools	14
1.3	Road Map	15
	To Learn Rather Than Do	17
1.4	Typography Conventions	18
2	General Use	19
2.1	Pre-Optimization	19
	Loading the Software	19
	Asset Names	19
	Prices and Other Imports	19
	Variance Matrix	20
	Adding a Benchmark to the Variance	22
	Expected Returns	22
2.2	The Optimization	23
2.3	Post-Optimization	23
	Explore the Trade	23
	Export the Trade	24
3	Optimizing Long-Only Portfolios	27
3.1	Required Inputs	27
3.2	Units of the Inputs	28
3.3	Examples for Passive Portfolios	28
	Minimize the Variance of the Portfolio	28
	Minimize Tracking Error	28
3.4	Examples for Active Portfolios	30
	Maximize the Information Ratio	30
	Maximize the Information Ratio with a Tracking Error Constraint	30
	Mean-Variance Optimization	32
	Buy-Hold-Sell List	32
3.5	Managing Cash Flow	33
	Injecting Money into a Portfolio	33
	Extracting Money out of a Portfolio	34
3.6	Going Farther	34

4	Optimizing Long-Short Portfolios	35
4.1	Required Inputs	35
4.2	Units of the Inputs	36
4.3	Examples	36
	Maximize the Information Ratio	36
	Maximize Return with a Bound on the Variance	37
	Minimize Variance Given a Long List and a Short List	38
	Mean-Variance Optimization	39
4.4	Managing Cash Flow	39
	Injecting Money into a Portfolio	39
	Extracting Money out of a Portfolio	40
4.5	Market Neutrality	40
4.6	Pairs Trading	40
4.7	Money Constraints	42
4.8	Real-Time Monitoring	43
4.9	Benchmarks	44
4.10	Going Farther	44
5	Asset Allocation	47
5.1	Efficient Frontiers	47
5.2	Examples	48
	Simple Three Variable Problem	49
	Zero Variance Cash	50
5.3	Asset Allocation with a Benchmark	51
	The Mathematics	53
5.4	Asset Allocation via the Portfolio Optimizer	53
5.5	Simultaneous Multiple Scenario Analysis	54
	Min-Max Solution	57
	General Simultaneous Optimization	58
5.6	Going Farther	59
6	Trading Costs	61
6.1	Background	61
6.2	Specifying Costs	61
	Linear Costs	62
	Polynomial Costs	63
	Other Nonlinear Costs	64
6.3	Power Laws	65
6.4	On Scaling Costs and Expected Returns	65
6.5	Costs Due to Taxes	66
6.6	Trade-Dependent Costs	66
6.7	Going Farther	67
7	Constraints	69
7.1	Summary of All Constraints	69
7.2	Integer Constraints	71
	Number of Assets to Trade	71
	Number of Assets in the Portfolio	71
	Number of Positions to Close	71
	Round Lots	72

7.3	Constraining Turnover	72
7.4	Lower and Upper Trading Bounds	72
	Liquidity Constraints	72
7.5	Simple Weight Constraints	74
	Asset Allocation	74
	Portfolio Optimization	74
7.6	Sums of Largest Weights	75
7.7	Linear Constraints	76
	Building Constraints	77
	Constraints on Trades and/or Absolute Values	78
	Looking at the Effect of the Constraints	79
	Evaluating Un-imposed Constraints	80
7.8	Threshold Constraints	81
	Trade Thresholds	81
	Portfolio Thresholds	81
	Summary of Threshold Inputs	82
7.9	Constraints on Variances, Tracking Errors and Expected Returns	82
	Variance Constraints	82
	Benchmark (Tracking Error) Constraints	83
	Alpha (Expected Return) Constraints	84
7.10	Forced Trades	84
7.11	Cost Constraints	84
7.12	Constraint Penalties and Soft Constraints	85
7.13	Quadratic Constraints	86
7.14	Going Farther	87
8	Generating Random Portfolios	89
8.1	Uses	89
	Fund Manager Performance	90
	The Effect of Constraints	90
	The Efficacy of Returns Prediction	90
	The Basis of an Investment Mandate	90
	Opportunity Analysis	91
	The Quality of a Risk Model	91
	Portfolio Construction Process Attribution	91
	Benchmarking Mandate Breaches	91
	Bidding on Portfolios	91
	Optimizing a Non-standard Utility	91
8.2	The Command	91
	Exporting Random Portfolios	93
	Working with Random Portfolios	94
8.3	Example: Exploring Alpha Generation	94
8.4	Example: Bidding on an Unknown Portfolio	98
8.5	Going Farther	103
9	Practicalities and Troubleshooting	105
9.1	Easy Ways to Get the Optimization Wrong	105
	Data Mangling	105
	Input Mangling	106
9.2	Special Optimizations	107

Considering Higher Moments	108
Minimize Costs	109
Non-standard Utilities	110
9.3 Troubleshooting	110
Utility Problems	110
Portfolio Problems	110
Miscellaneous Problems	111
10 Adjusting Speed and Quality	113
10.1 Staying at a Given Solution	113
10.2 Reducing Time Use	114
10.3 Improving Quality	114
11 Some Philosophy	117
11.1 The Textbook View	117
What's wrong with the textbook view?	117
What's been the harm?	117
What can be salvaged?	118
11.2 Investor Optimality	118
11.3 Bayesian Statistics	118
11.4 Generating Alpha	119
11.5 Noisy Inputs	119
Resampling Methods	119
Increase Risk Aversion	120
Increase Trading Costs	120
12 Advanced Features	121
12.1 Dual Benchmarks	121
12.2 Multiple Variances and Expected Returns	123
Rival Forecasts	123
Multiple Time Periods	125
Credit Risk	125
Scenarios	126
12.3 Suppressing Warning Messages	127
Portfolio Optimizer Warnings	127
Asset Allocator Warnings	128
12.4 Compact Variance Objects	128
The Variance List	129
13 Computational Details	131
13.1 POP Constituents	131
13.2 The Objectives	133
The Utilities	133
Constraints	134
13.3 Approximate Optimality	134
13.4 Multiple Variances and Expected Returns	135
The Variance and Alpha Tables	135
The Utility Table	136
Controlling the Objective	138
13.5 The Genetic Algorithm	138

Algorithm Components	139
Procedure	139
Differences Between Asset Allocation and Portfolio Optimization	140
13.6 Single Trade Optimization	140
13.7 Writing C or C++ Code	140
13.8 Bug Reporting	141

List of Tables

3.1	Units of arguments for long-only portfolio optimization.	28
4.1	Units of arguments for long-short portfolio optimization.	36
7.1	Asset allocation constraints.	69
7.2	Portfolio optimization constraints.	70
8.1	S objects for alpha generation example.	97
13.1	Categorization of S functions in POP.	132
13.2	Arguments for multiple variances and expected returns.	136

List of Figures

1.1	Routes through the document for four tasks.	16
4.1	Constraints on gross, net, long and short values.	43
5.1	An efficient frontier.	48
5.2	Four variable asset allocation.	51
5.3	Efficient frontiers under three scenarios.	56
8.1	Optimal portfolio return relative to random returns.	97
8.2	Outperformance of random period by period.	99
8.3	Distribution of biggest assets in random portfolios.	102

Chapter 1

Introduction

This chapter has diverse aims:

- It discusses the role of portfolio optimization, and its neglect.
- It explains what software you need in order to run POP.
- Given one of five tasks, it suggests a route through the rest of this document.
- It presents the typographic conventions of the document.

1.1 The Role of Portfolio Optimization

Optimization for active portfolios combines the intuition and analysis of the fund manager with a variance matrix to create a portfolio that takes advantage of the manager's information while keeping risk minimal. Reducing risk makes the manager's predictions more valuable.

Optimization is not used as much as it might be. Historically there has been a lack of adequate tools. Inertia and a dearth of practical advice play a part as well.

Some reasons for resistance to optimization are the beliefs:

1. You need to be quantitative to do optimization.
2. Optimizations are too complicated to do.
3. Optimizations yield wild solutions.

These are more fears than facts:

1. If you are quantitative enough to use a spreadsheet, you are quantitative enough to perform optimizations with POP. The requirements of the optimization are a variance matrix and, often, expected returns for the assets. There is a function in the package that will build a variance matrix—you don't especially need to know what a variance matrix is (it's the appropriate mathematical package of volatilities and correlations). If you have

views on the assets in your universe, then you are close to having expected returns for them.

The key issues in optimization are financial, not quantitative.

2. Some optimizers *are* complex to run. Effort has been taken with POP to make simple optimizations as easy as possible. Inputs and results are in terms that are familiar and useful—amounts of money and units of the assets (rather than weights). POP will do some very complicated optimizations and hence those will be relatively involved to do. However, you can ignore complications until you need the functionality.
3. Portfolio optimization has received some very bad press. There have been many statements about wild results. This actually is true if you put no controls on it, as many textbooks present the subject (see Section 11.1 on page 117). Automobiles have not been abandoned because they need to be controlled, and neither should portfolio optimization. If you keep your hand on the wheel, then optimization can take you to a better portfolio. It is not unreasonable that you should use relatively tight constraints until you become more comfortable with the process.

A benefit of optimizing is that it can force you to think more clearly about your investment process. Investment is balancing opportunity with risk and cost. Even if your initial optimizations do not capture the balance well, the issues are likely to be more clear to you.

While optimization is a valuable tool, it is not a magic bullet. Results are not guaranteed. Because of statistical noise in the estimates of the expected returns and the variance, actual results are on average worse than those that optimizations suggest.

With the availability of random portfolios in POP it is now possible to test the effect of constraints. Previously constraints had to be set via intuition and superstition. Now the constraints can be examined critically, and set with some measure of rationality. This means that optimization can be of even more value.

1.2 Necessary Tools

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

- R, which can be downloaded for free via:
<http://www.r-project.org/>
- S-PLUS, sold by Insightful:
<http://www.insightful.com/>
- C or C++. You can call POP optimizers in a program that you write.

The last option of using C is not recommended—it requires considerable effort, and likely has little or no benefit.

S-PLUS and R are versions of the S language, and POP has been written to work with either version of S. This document assumes you are using S (as opposed to using C code).

When this document says “S”, it means either R or S-PLUS—the term “S” should not be construed to mean only S-PLUS.

Programming experience is not mandatory—whatever your objective, there is likely to be an example provided in this manual that fits your case.

The present document assumes knowledge of S to the level of “A Guide for the Unwilling S User”, a brief introduction which can be found in the Tutorials section of <http://www.burns-stat.com/>. Commands beyond that level are included and explained.

Performing the optimizations and using the results requires very little involvement with S. An exception is generating random portfolios where the portfolios probably need to be manipulated. For this reason Chapter 8 has more extensive examples of using S.

While it is reasonably easy to start using POP, there is a lot of room to grow. POP’s flexibility and the power of S means that your creativity can carry you a long way.

1.3 Road Map

We divide POP’s capabilities into four tasks, and suggest a different route through the document for each one. Figure 1.1 is a graphical view of the suggestions presented below.

The routes can be divided into three steps.

Step I First read Chapter 2 to get an overview. This covers preparation for the optimization, and what to do once the optimization has been performed.

Step II This step depends on which of the four tasks you are doing:

- **Long-Only Optimization**

Read Chapter 3. There is a set of examples for passive portfolios, and another set for active portfolios. Before you act on an optimization, it would be best to consult Section 9.1 on page 105 to try to avoid common mistakes.

Chapter 6 on costs, Chapter 7 on constraints, and Chapter 10 on speed and quality should then be consulted if pertinent.

- **Long-Short Optimization**

Read Chapter 4. Before you act on an optimization, it would be best to consult Section 9.1 on page 105 to try to avoid common mistakes.

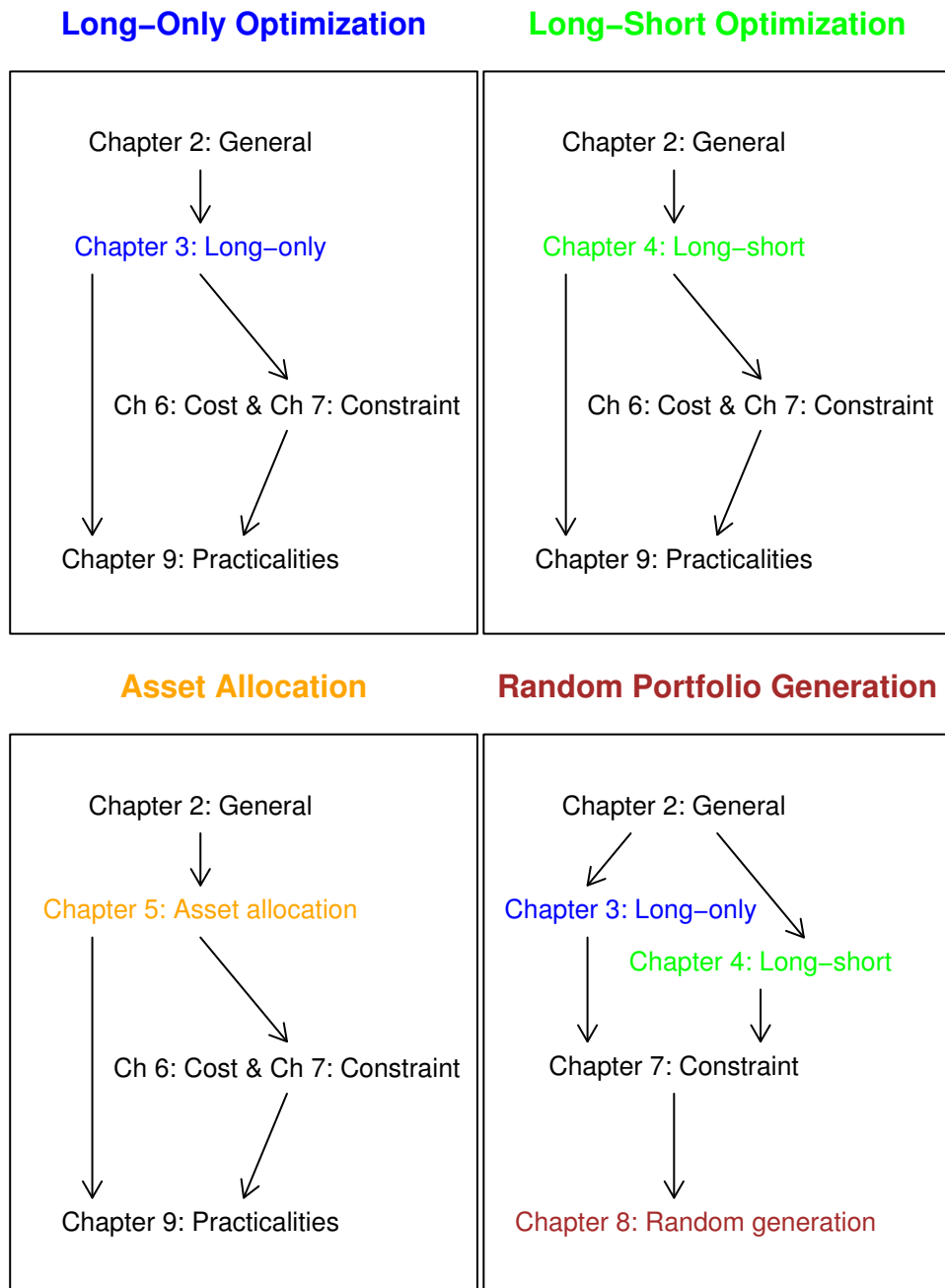
Chapter 6 on costs, Chapter 7 on constraints, and Chapter 10 on speed and quality should then be consulted if pertinent.

- **Asset Allocation**

Read Chapter 5. Before you act on an optimization, it would be best to consult Section 9.1 on page 105 to try to avoid common mistakes.

Chapter 6 on costs, Chapter 7 on constraints, and Chapter 10 on speed and quality should then be consulted if pertinent.

Figure 1.1: Routes through the document for four tasks.



- **Random Portfolio Generation**

First you may want to browse through Chapter 3 or Chapter 4 depending on whether your portfolios are long-only or long-short. Then read Chapter 7 on constraints. Next read Chapter 8 on the random generation itself.

Before you act on a computation, it would be best to consult Section 9.1 on page 105 to try to avoid common mistakes.

Step III An optional step if and when you want to go beyond the basics.

Some general issues on optimization are discussed in Chapter 11. Chapter 12 covers advanced features of the optimizers. Finally, Chapter 13 explains computational issues of the optimization.

Chapters 12 and 13 are of a more technical nature, and need not normally be understood to use the optimizers.

To Learn Rather Than Do

If you merely want to learn about portfolio construction, then Chapter 11 is the most obvious target. Browsing through the rest of the manual is advised. Special attention might be given to:

- Section 1.1 on the role of portfolio optimization.
- Section 4.5 on market neutrality.
- Section 5.5 on performing simultaneous scenario estimation.
- Section 6.3 on the square root rule for trading costs.
- Section 6.4 on the scaling of costs.
- Section 6.5 on taxes as costs.
- Section 8.1 on the uses of random portfolios.
- Section 9.1 on common mistakes that are made.
- Section 9.2 on higher moments.
- Section 11.5 on noisy inputs.
- Section 12.1 on dual benchmarks.
- Section 12.2 on multiple variances and expected returns.
- Section 13.2 on the objectives of the optimizers.
- Section 13.3 on approximate optimality.

1.4 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 <- portfolio.optimizer(prices, varian, gross.val=1e6,
+   long.only=T)
```

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 boxes containing material of an advanced nature. The advanced material should generally be skipped initially.

Chapter 2

General Use

This chapter briefly surveys POP's functionality. More detailed description of the optimization process itself is reserved for later chapters.

2.1 Pre-Optimization

Loading the Software

If POP was installed in the default place, then POP is loaded into an R session on Windows with:

```
library(POP.R, lib.loc='C:/Program Files/BurSt/POP')
```

An analogous statement is used under Linux. In S-PLUS it will depend on the particular installation.

Asset Names

The `portfolio.optimizer` function demands some of its inputs to have asset names. While `asset allocator` allows you to give objects with no asset names at all, it is good practice to use asset names. The functions do name matching. If there are names on all of the objects, then the optimizers will make sure that objects are properly aligned. Names also make it easier to check the validity of the inputs.

Without asset names on some inputs, warnings will be issued that the correct ordering of the assets within the object is being assumed.

When there are asset names, many inputs to the optimizers can include extra assets—the extraneous assets are removed before the actual computations proceed. The `prices` argument controls the assets used in `portfolio.optimizer` and `random.portfolio`, while the `expected.return` argument controls the assets used in `asset allocator` and `efficient.frontier`.

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 needed when there is a single column (or row) of data.

Variance Matrix

A variance matrix is always necessary for optimization in POP. 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 will probably be more appropriate.

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

```
> retmat <- diff(log(price.history))
> varian <- factor.model.stat(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 prices vector that is given to the portfolio optimizer 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; etcetera.

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 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 (eventually at least) be a factor when used as a 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 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="fact")
```

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 `var.add.benchmark` function:

```
> varian <- var.add.benchmark(varian.assets, spx.weights,
+ "spx")
```

This function takes three arguments—a variance matrix, a vector of weights (that should sum to 1), and a character string giving the name for the benchmark. The weight vector needs to have names which are (some of) the asset names.

The computation performed by this function is the preferred method of introducing a benchmark into a variance matrix for optimization. See [Burns, 2003b] for a study on approaches to incorporating benchmarks into the variance matrix for optimization.

Expected Returns

If the portfolio is active, then expected returns for the assets generally need to be estimated as well as a variance matrix.

It is often the case that you will have a ranking of the assets into some number of categories. You then need to turn these rankings into expected returns before the optimization can proceed. There are at least two approaches to performing this transformation:

- If you have past data, you can backtest strategies of how to change the rankings into expected returns.
- You can try a number of schemes, and see which are the ones that produce the most appealing optimal portfolios.

Obviously backtesting can provide a measure of assurance. The second method, though, may lead to a re-evaluation of the ranking scheme. You may, for instance, decide that not all rank “4” assets are really the same—that perhaps the ranking scheme should have more levels. [Chriss and Almgren, 2005] provide a theoretically optimal technique for transforming rankings into expected returns.

You can do optimizations without expected returns by having buy, sell and hold lists. The good thing about this is that it is very easy to do. However, it really gives expected returns to the assets in an uncontrolled manner. Since all “buy” assets are treated equally, more volatile stocks are implicitly given larger expected returns than less volatile stocks with similar correlations.

2.2 The Optimization

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

- Long-only portfolio optimization (either active or passive): Chapter 3.
- Long-short portfolio optimization: Chapter 4.
- Asset allocation: Chapter 5.
- Random portfolio generation: Chapter 8.

Trading costs are discussed in Chapter 6, while Chapter 7 covers constraints.

Chapters 9 and 12 have additional information—Section 9.1 on page 105 is particularly recommended.

caution

If 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 105 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, should at least be fairly reasonable.

2.3 Post-Optimization

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

S code note

S, like many other computer languages, accepts exponential notation for numbers. Thus `1e6` means one-million—a one followed by six zeros. 21 basis points (0.0021) could be written as either `21e-4` or `2.1e-3`.

The `summary` function provides information on:

- the utility, costs, and penalties for broken constraints
- opening and closing positions
- valuations of the portfolio and trade (not done for asset allocation)

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)
```

You can also get the valuation of the individual assets as well as overall valuation using the `valuation` function:

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

The default behavior of `valuation` when given a portfolio 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, the prices must be given:

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

Prices are optional when portfolio objects are given—the prices used during the optimization are used if none are specified.

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.

Chapter 3

Optimizing Long-Only Portfolios

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

3.1 Required Inputs

If you are building or changing a long-only portfolio, then you need to set the `long.only` argument of `portfolio.optimizer` to `TRUE`. The optimizer always requires a vector of prices and a variance matrix.

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 constraint matrix—may contain additional assets. Assets not in `prices` will be ignored. However, the other objects must contain all of the assets that are in `prices`.

Trading is forced to be in integer amounts except when closing out a non-integral existing position. If you want round lotting performed, you should give prices and other quantities in terms of lots.

You also need to indicate the desired amount of money in the final portfolio, or the maximum amount of money to trade. State the amount of money in the final portfolio by giving the `gross.value` argument. This can either be a single number or two numbers which give an allowable range. 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.

In general the optimizer has no problem with a constraint this tight—it most likely can be tighter if that is important to you. However, if the portfolio is small relative to the prices of the assets, the utility may suffer from a tight constraint even if the value constraint is met.

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.

The alternative to stating the final value of the portfolio is to give an upper limit on the amount of money to trade. To do this, give the `trade.value`

Table 3.1: Units of arguments for long-only portfolio optimization.

currency units per asset unit	asset units	currency units
prices	existing	gross.value
long.buy.cost	lower.trade	trade.value
long.sell.cost	upper.trade	bounds.constraint
	start.sol	

argument the gross amount (buys plus sells) that you want to allow.

You can give both the final value and the maximum amount to trade.

3.2 Units of the Inputs

Many of the inputs are in units of the currency (e.g., dollars), units of the assets (e.g., shares or lots), or units of currency per unit of asset. Table 3.1 displays the arguments of `portfolio.optimizer` that are useful for long-only portfolios in each of these categories.

The `max.weight` and `sum.weight` arguments are in fractions of the gross value.

3.3 Examples for Passive Portfolios

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

Minimize the Variance of the Portfolio

This is the simplest optimization to perform:

```
> op.mv1 <- portfolio.optimizer(prices, varian, long.only=T,
+   gross.val=1.4e6)
```

The amount put into the portfolio is 1.4 million currency units. The simplicity of this command means that it probably is not what you want since it is dependent on default values for some important arguments. The `op.mv1` object will represent a minimum variance portfolio containing up to 20 assets (20 is the default for `ntrade`).

It is unusual to minimize the absolute variance of the portfolio. However, see [Burns, 2003c] and [Burns, 2003d] for a discussion of hyperpassive funds. These are passive funds that minimize portfolio variance.

Minimize Tracking Error

More common than minimizing variance is to 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 needs to be in the variance matrix. If it isn't and you have the weights of the constituents, refer to page 22. If the benchmark is in the variance matrix, then you may want to consult [Burns, 2003b] to confirm that the benchmark was added in a suitable way.

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 <- portfolio.optimizer(prices, varian, long.only=T,
+   gross.val=1.4e6, benchmark="spx")
```

This is a very simple command, so is unlikely to be exactly what you want.

If you are building a new portfolio, then you want to give the `ntrade` argument the number of assets that you want in the portfolio:

```
> op.te2 <- portfolio.optimizer(prices, varian, long.only=T,
+   gross.val=1.4e6, benchmark="spx", ntrade=55,
+   max.weight=0.05)
```

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%.

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

```
> op.te3 <- portfolio.optimizer(prices, varian, long.only=T,
+   gross.val=1.4e6, benchmark="spx", max.weight=0.05,
+   existing=cur.port)
```

The command creating `op.te3` is allowing up to 20 assets to be traded (the default value for `ntrade`). It does not control the turnover, nor the number of assets in the resulting portfolio. Thus `op.te3` 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.te4 <- portfolio.optimizer(prices, varian, long.only=T,
+   gross.val=1.4e6, benchmark="spx", max.weight=0.05,
+   existing=cur.port, trade.val=1e5, port.size=c(45,55))
```

This command is (finally) a realistic possibility. You might want to adjust the `ntrade` argument from its default value of 20.

See Section 3.5 on page 33 for examples of dealing with cash flow.

3.4 Examples for Active Portfolios

Expected returns of the assets are required for optimization of active portfolios. (This is not absolutely true—see page 32 for an example.) Additionally, there needs to be a decision on how to balance expected return with risk.

Maximize the Information Ratio

The easiest optimization that involves expected returns 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 to maximize—we're seeking the best risk-adjusted return.

A command to maximize the information ratio is:

```
> op.ir1 <- portfolio.optimizer(prices, varian, long.only=T,
+   expected.ret=alphas, gross.val=1.4e6)
```

The `alphas` object is a vector of expected returns of the assets. When 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.val * 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 more realistic case of maximizing the information ratio when there is a benchmark involved.

Maximize the Information Ratio with a Tracking Error Constraint

Currently it is common practice 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 smaller than a given value. See [Burns, 2003c] and [Burns, 2003d] for more detailed arguments on this point. Also [Burns, 2004] comments on tracking error.

The mechanism in `portfolio.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 <- portfolio.optimizer(prices, varian, long.only=T,
+   expected.ret=alphas, gross.val=1.4e6,
+   bench.constraint = c(spx=0.04^2/252))
```

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 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—similar to the traditional practice that is less satisfactory.

A more realistic command that builds a new portfolio maximizing the information ratio with a limit on the tracking error is:

```
> op.ir3 <- portfolio.optimizer(prices, varian, long.only=T,
+   expected.ret=alphas, gross.val=1.4e6, ntrade=55,
+   bench.constraint = c(spx=0.04^2/252), max.weight=.05)
```

To update an existing portfolio, you might issue a command similar to:

```
> op.ir4 <- portfolio.optimizer(prices, varian, long.only=T,
+   expected.ret=alphas, gross.val=1.4e6, ntrade=15,
+   bench.constraint = c(spx=0.04^2/252), max.weight=.05,
+   exist=cur.port, trade.val=1e5, port.size=60)
```

This command has a limit on the turnover and specifies that the final portfolio should contain up to 60 assets.

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

```
> op.ir5 <- portfolio.optimizer(prices, varian, long.only=T,
+   expected.ret=alphas, gross.val=1.4e6, ntrade=15,
+   bench.constraint = c(spx=0.04^2/252, newspx=0.05^2/252),
+   max.weight=.05, exist=cur.port, trade.val=1e5,
+   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 in Section 12.1 on page 121.

Mean-Variance Optimization

The default objective is to maximize an information ratio whenever expected returns are given. If you want to perform a mean-variance optimization instead, then you need to specify the `objective` 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 134 has more on the risk aversion parameter.

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

```
> op.mv1 <- portfolio.optimizer(prices, varian, long.only=T,
+   expected.ret=alphas, gross.val=1.4e6,
+   objective='mean-var', risk.aver=.7)
```

This command does not include a benchmark, so is probably not what you want. If you do have a benchmark, a good approach to take is to do the optimization in the absolute sense (as in `op.mv1`) with a constraint on the tracking error from the benchmark. The tracking error constraint is placed on the optimization with a command like:

```
> op.mv2 <- portfolio.optimizer(prices, varian, long.only=T,
+   expected.ret=alphas, gross.val=1.4e6,
+   objective='mean-var', risk.aver=.7,
+   bench.constraint = c(spx=0.04^2/252))
```

For an explanation of the value given for `bench.constraint`, see page 31.

In mean-variance optimization, the risk aversion parameter is quite important. It may take some experimentation to find the risk aversion that gives you an optimization that you like.

Buy-Hold-Sell List

Though it is recommended, it is not absolutely necessary to create expected returns. 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 <- portfolio.optimizer(prices, varian, long.only=T,
+   gross.val=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 `trade.value` argument to constrain turnover, and the `port.size` argument to constraint the number of names in the optimized portfolio.

3.5 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 substantial cash flow, then this process has the added 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 <- portfolio.optimizer(prices, varian, long.only=T,
+   expected.ret=alphas, trade.val=newcash, ntrade=5,
+   bench.constraint = c(spx=0.04^2/252), max.weight=.05,
+   exist=cur.port, lower.trade=0)
```

The key features in this command are setting the `trade.value` 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 <- portfolio.optimizer(prices, varian, long.only=T,
+   expected.ret=alphas, trade.val=cash.required, ntrade=5,
+   bench.constraint = c(spx=0.04^2/252), max.weight=.05,
+   exist=cur.port, upper.trade=0)
```

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

3.6 Going Farther

Tasks that you might want to undertake:

- To add trading costs to your optimization command, see Chapter 6 on page 61.
- To add constraints, see Chapter 7 on page 69.
- To review common mistakes, see Section 9.1 on page 105.
- To improve your solution or examine the quality of solutions you are getting, go to Chapter 10 on page 113.
- To export your solution to a file, see Section 2.3 on page 23.
- To perform optimizations with multiple variances, go to Section 12.2 on page 123.
- To optimize over multiple benchmarks, read Section 12.1 on page 121.

Chapter 4

Optimizing Long-Short Portfolios

This chapter discusses commands to optimize long-short portfolios.

4.1 Required Inputs

The `portfolio.optimizer` function needs a price vector and a variance matrix.

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 constraint matrix—may contain additional assets (which will be ignored).

Trading is forced to be in integer amounts except when closing out a non-integral existing position. If you want round lotting performed, you should give prices and other quantities in terms of lots.

The optimizer also requires the monetary value of the portfolio or the maximum amount to trade. The arguments that control the value of the portfolio and the trade are `gross.value`, `net.value`, `long.value`, `short.value` and `trade.value`. These are all in units of the currency.

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 `portfolio.optimizer` takes the absolute value of negative numbers 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. The trade value is the sum of the buys and sells in the trade—similar to the gross value for the portfolio.

There are three minimal sets of these arguments:

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

You can add others of these arguments if you wish. For example, giving `gross.value`, `net.value` and `trade.value` is a likely choice. But if any of

Table 4.1: Units of arguments for long-short portfolio optimization.

currency units per asset unit	asset units	currency units
prices	existing	gross.value
long.buy.cost	lower.trade	net.value
long.sell.cost	upper.trade	long.value
short.buy.cost	start.sol	short.value
short.sell.cost		trade.value
		bounds.constraint

the gross, net, long or short are given, then there needs to be at least one pair of them given (either gross and net, or long and short).

The four arguments that control the value of the portfolio: `gross.value`, `net.value`, `long.value`, `short.value` are logically each of length two—giving an allowable range. 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. However, `net.value` equal to zero is a special case and may produce an acceptable range for you. Section 4.7 on page 42 gives more details about constraining the value of the portfolio.

4.2 Units of the Inputs

Many of the inputs are in units of the currency (e.g., dollars), units of the assets (shares or lots for example), or units of currency per unit of asset. Table 4.1 displays the arguments of `portfolio.optimizer` in each of these categories.

The `max.weight` and `sum.weight` arguments are in fractions of the gross value.

4.3 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 6). Nor does it cover several of the constraints (see Chapter 7).

Maximize the Information Ratio

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

```
> opt.ir1 <- portfolio.optimizer(prices, varian,
+   expected.ret=alphas, gross.val=1.4e6,
+   net.val=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 <- portfolio.optimizer(prices, varian,
+   expected.ret=alphas, gross.val=1.4e6,
+   net.val=c(-1e4, 2e4), ntrade=55, max.weight=.05)
```

This new command adds `ntrade` to give the maximum number of assets to be traded—so this will create 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 <- portfolio.optimizer(prices, varian,
+   expected.ret=alphas, gross.val=1.4e6,
+   net.val=c(-1e4, 2e4), ntrade=15, max.weight=.05,
+   existing=cur.port, trade.val=5e4)
```

Here both the number of assets in the trade and the turnover are limited, and of course the existing portfolio is specified. An additional argument that may be of interest is `port.size` which controls the number of assets in the portfolio.

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 <- portfolio.optimizer(prices, varian,
+   expected.ret=alphas, gross.val=1.4e6,
+   net.val=c(-1e4, 2e4), ntrade=55, max.weight=.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 should add a couple more arguments:

```
> opt.vc2 <- portfolio.optimizer(prices, varian,
+   expected.ret=alphas, gross.val=1.4e6,
+   net.val=c(-1e4, 2e4), ntrade=55, max.weight=.05,
+   var.constraint=.04^2/252, objective='mean-var',
+   risk.aver=0)
```

This new command now does mean-variance optimization with a risk aversion parameter of zero—that is, ignore the variance (except for the constraint) and maximize the 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 <- portfolio.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)
```

The `lower.trade` and `upper.trade` arguments to `portfolio.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.

If you want to allow all of the assets in the short and long lists to be in the portfolio, then you can set the `ntrade` argument to some large number. The optimizer will adjust it to reflect the number of assets that can trade.

This example is a simple case—the lower and upper bounds can be more complicated when there is an existing portfolio. When a portfolio is already in place, the calculations are similar to those starting on page 72 for liquidity constraints.

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 the `forced.trade` argument.

```

> opt.list2 <- portfolio.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

Mean-variance optimization maximizes the expected return of the portfolio minus the risk aversion parameter times the variance of the portfolio. More details of the risk aversion parameter are on page 134.

To do mean-variance optimization, you need to give expected returns and set the objective argument to 'mean-var'. You will almost surely need to specify the risk aversion as well. Typically it takes some experimentation to get the risk aversion that matches your intuition.

```

> opt.mv1 <- portfolio.optimizer(prices, varian,
+   expected.ret=alphas, gross.val=1.4e6,
+   net.val=c(-1e4, 2e4), ntrade=55, max.weight=.05,
+   objective='mean-var', risk.aver=3.7)

```

4.4 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 change the balance between longs and shorts. Similar commands would be used to achieve this.

Injecting Money into a Portfolio

If you want to perform optimization with a constraint on the variance (see page 37), then a command that will do this while only increasing existing positions is:

```

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

```

The key points here are setting the gross value to the desired new amount, and forcing the trade value to equal the amount by which the gross is increased. 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 `trade.value` 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 <- portfolio.optimizer(prices, varian,
+   expected.ret=alphas, gross.val=curgross - cash.require,
+   net.val=c(-1e4, 2e4), ntrade=15, max.weight=.05,
+   var.constraint=.04^2/252, existing=cur.port,
+   trade.val=cash.require, universe.trade=names(cur.port))
```

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

4.5 Market Neutrality

A goal that many long-short portfolios have is to be market neutral. That is, to have zero correlation between the portfolio and some representation of the “market”.

[Amenc and Martellini, 2002] report hedge fund data where the market neutral style has one of the higher correlations with the market among styles of hedge funds. This can partly be explained by long-short funds being classified as market neutral whether or not they claim to be.

However, funds that attempt to be market neutral do not always succeed very well. In particular, being dollar neutral—having a net value near zero—is exceptionally ineffective at achieving market neutrality.

[Burns, 2003a] provides some perspective on market neutrality and some hints on how to achieve it.

4.6 Pairs Trading

One strategy of building long-short portfolios is to only trade specific pairs of assets. Here we show an approach to setting up the constraints so that trading is restricted to a specified set of pairs.

We start by defining a function that creates the desired constraints (see Section 7.7 for an explanation of the use of linear constraints). It takes an argument called `pairmat` which is meant to be a two-column matrix of asset names. Each row of `pairmat` contains the names of the assets in one pair. Thus `pairmat` will have as many rows as there are pairs to be traded. The other argument to the function is `prices` which should be the same vector of prices as is used in the call to the optimizer.

```
> pair.constraint <- function(pairmat, prices)
+ {
+   nr <- nrow(pairmat)
+   if(nr < 1) stop("bad value for pairmat")
+   amat <- array(0, c(length(prices), nr),
+     list(names(prices), apply(pairmat, 1, paste,
```

```

+         collapse="-"))
+   bound <- numeric(nr)
+   for(i in 1:nr) {
+     amat[pairmat[i,], i] <- 1
+     bound[i] <- min(prices[pairmat[i,]])
+   }
+   bound <- .9 * bound
+   boundmat <- cbind(-bound, bound)
+   dimnames(boundmat) <- list(dimnames(amat)[[2]], NULL)
+   list(matrix=amat, bounds=boundmat)
+ }

```

S code note

The `array` function creates matrices (which are two-dimensional arrays) and higher dimensional arrays.

The `apply` function performs some operation on each row or column of a matrix. In this case the `paste` function is applied to each row of the matrix. An additional argument to `paste`, `collapse`, is given in the call to `apply`. You can see the result of this operation in the `dimnames` of the components of `jjpc` below.

The result of the function is a list containing two matrices—one containing the constraint matrix and the other containing the bounds. There is one constraint for each pair. In the constraint matrix there are ones corresponding to the assets that are in the pair and zeros everywhere else. The bounds are set to be slightly smaller than the smaller of the two prices in the pair.

This function is illustrated with a toy example:

```

> jjpm
  [,1] [,2]
[1,] "C" "H"
[2,] "E" "F"
[3,] "A" "G"
> qats.price10
  A      B      C      D      E      F      G      H      I      J
27.63 19.46 11.67  5.79  5.15 20.99 12.07  8.13  7.27  5.28
> jjpc <- pair.constraint(jjpm, qats.price10)
> jjpc
$matrix
  C-H E-F A-G
A  0  0  1
B  0  0  0
C  1  0  0
D  0  0  0
E  0  1  0
F  0  1  0
G  0  0  1
H  1  0  0
I  0  0  0
J  0  0  0

```

```

$bounds
      [,1] [,2]
C-H -7.317  7.317
E-F -4.635  4.635
A-G -10.863 10.863

```

In words, we are forcing the long position of one of the pair to be almost equal to the short position of the other. Since the net value within a pair can be adjusted to be as close to zero as about half the price of the less expensive of the pair, we set the bounds to be a little larger than that.

Once the constraints have been created, we can use them in an optimization:

```

> jjop <- portfolio.optimizer(qats.price10, qats.var10,
+   qats.alpha10, gross.value=1e4, net.value=c(-Inf, Inf),
+   constraint.mat=jjpc$mat, bounds=jjpc$bound,
+   universe.trade=jjpm)

```

The key items here are the `constraint.matrix` and `bounds.constraint` arguments. We are not constraining the net value at all since the constraints are already doing this.

The other thing that needs to be done is to restrict trading to only assets that are in a pair. One way of doing that is to give prices only for assets that are in a pair. The approach taken here is to give the `universe.trade` argument the names that can be traded (the matrixness of `jjpm` is ignored in this case).

```

> jjop$trade
  A   C   E   F   G   H
46 -161 359 -88 -105 232

```

Since the prices will generally change each time the optimization is done, the `pair.constraint` function should probably be run each time in order to make sure that the bounds are properly set.

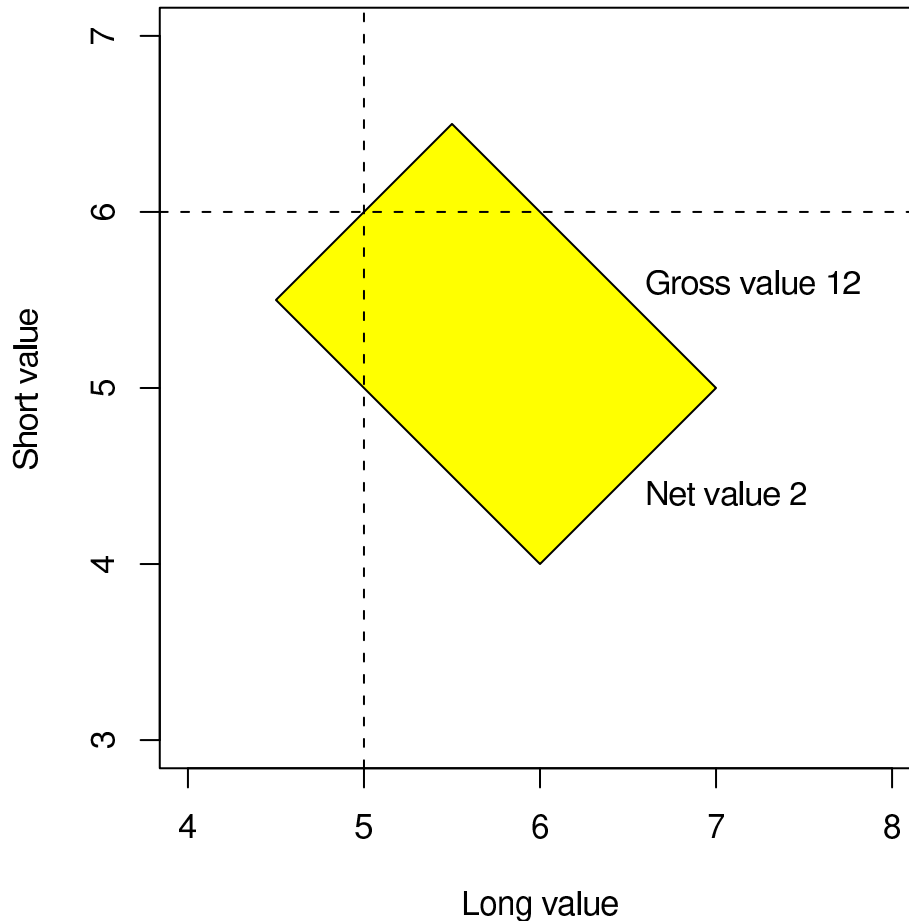
4.7 Money Constraints

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. The quality of the optimization may influence the amount of leverage desired, in which case a new optimization would be undertaken with revised values for the gross and net. Remember though, that the expected return and the volatility found for the optimal portfolio are biased towards optimism.

The constraints on the long and short values are not just redundant to the gross and net value constraints. Figure 4.1 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

Figure 4.1: Constraints on gross, net, long and short values.



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.

4.8 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
```

```

+ cur.port <- update.portfolio() # user-defined function
+ cur.opt <- portfolio.optimizer(prices, varian,
+   expected.ret=alphas, trade.value=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.

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.

4.9 Benchmarks

caution

The `benchmark` and `bench.constraint` arguments to `portfolio.optimizer` with long-short portfolios are not likely to produce the behavior that you expect.

In this package (and elsewhere) you are short the benchmark by the gross value of the portfolio. If that is your situation, you will be fine. If this is not the case, but you do have a benchmark—either in the traditional sense or there is a set of liabilities, then you can put the benchmark in the existing portfolio but not allow it to be traded.

4.10 Going Farther

Tasks that you might want to undertake:

- To add trading costs to your optimization command, see Chapter 6 on page 61.
- To add constraints, see Chapter 7 on page 69.
- To review common mistakes, see Section 9.1 on page 105.
- To improve your solution or examine the quality of solutions you are getting, go to Chapter 10 on page 113.

- To export your solution to a file, see Section 2.3 on page 23.
- To perform optimizations with multiple variances, go to Section 12.2 on page 123.

Chapter 5

Asset Allocation

Asset allocation is simpler than portfolio optimization. The most obvious difference is that the answer is a set of weights that sum to one, rather than quantities of assets. These weights may not be negative. Also in asset allocation there is no constraint on the number of weights that must be zero. The `asset allocator` function does not have provision for a benchmark (but see Section 5.3 on page 51 to see how to do this). Unlike in portfolio optimization, the default utility for asset allocation is mean-variance.

This chapter does not discuss trading costs or constraints—these topics are covered in Chapter 6 and Chapter 7.

5.1 Efficient Frontiers

The most likely start to an asset allocation problem is to compute efficient frontiers. An efficient frontier is created with the `efficient.frontier` function. This takes the arguments of `asset allocator` (plus an optional argument that controls how fine of resolution the frontier should be computed to).

A simple example of producing an efficient frontier is:

```
> ef1 <- efficient.frontier(aret, avar)
```

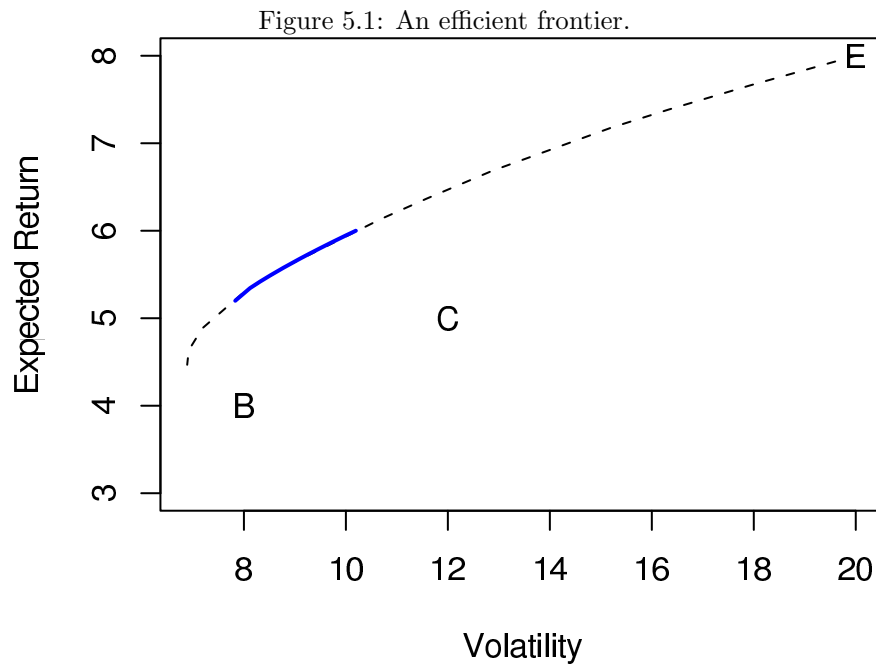
This merely passes a vector of expected returns and a variance matrix to the function. There is a `plot` method for the class of object that is returned, so a plot can be created with:

```
> plot(ef1)
```

More than one frontier can be put on the same plot. For example:

```
> ef2 <- efficient.frontier(aret, avar, min.weight=.1,  
+   max.weight=.5)  
> plot(ef2, add=T)
```

The second frontier has the constraints that all weights must be between 10% and 50%. Frontiers that are more constrained will generally be below less constrained frontiers.



Once a frontier is found that is suitable, then printing the efficient frontier object will indicate the risk aversion parameter that corresponds to your desired point on the frontier. The `asset allocator` function is then used to find the weight vector.

Figure 5.1 is produced with the following commands using data that are created in the next section.

```
> ef3 <- efficient.frontier(aret3, avar3)
> ef4 <- efficient.frontier(aret3, avar3, min.weight=.2,
+   max.weight=.4)
> plot(ef3, ylim=c(3,8), lty=2)
> plot(ef4, add=T, col="blue", lwd=2)
> text(avol3, aret3, c("E", "B", "C"))
```

The constraints in `ef4` make no difference in the location of the frontier in this case, but do significantly restrict its range. Since the extra constraints are merely on the ranges of weights, the solutions in `ef4` are also in `ef3`.

5.2 Examples

Here are some examples using `asset allocator`. To check your answer, you can verify that the mean and variance that are achieved in your allocation correspond to the point on the efficient frontier that you are aiming for.

Simple Three Variable Problem

We start with a problem involving three assets. First, let's create some data:

```
> anam3 <- c("equities", "bonds", "commodities")
> avol3 <- c(20, 8, 12)
> acor3 <- matrix(c(1, .2, .1, .2, 1, .1, .1, .1, 1),
+ 3, 3, dimnames=list(anam3, anam3))
> acor3
      equities bonds commodities
equities 1.0 0.2 0.1
bonds    0.2 1.0 0.1
commodities 0.1 0.1 1.0
> avar3 <- t(acor3 * avol3) * avol3
```

S code note

The command directly above transforms the correlation matrix into a variance matrix.

When a matrix is multiplied by a vector with length equal to the number of rows in the matrix, then each row of the matrix is multiplied by the corresponding element in the vector. This is the first thing done in the command, then the resulting matrix is transposed by the `t` function. The transposed matrix is then multiplied again by the vector of volatilities.

```
> avar3
      equities bonds commodities
equities 400 32.0 24.0
bonds    32 64.0 9.6
commodities 24 9.6 144.0
> aret3 <- c(equities=8, bonds=4, commodities=5)
```

note

These data (and subsequent data) are purely for demonstration purposes—they should not be taken as useful predictions.

A very simple call to `asset.allocator` is:

```
> aa3.1 <- asset.allocator(aret3, avar3, risk.aver=.01)
```

See page 134 on the scaling of the risk aversion parameter.

The optimal solution in this case is:

```
> round(aa3.1$new.portfolio * 100, 1)
      equities  bonds  commodities
      53.1      4.7      42.2
```

Here we have transformed the weights into percent and rounded. We may decide that we want no more than 45% in any one asset. The command to do that is:

```

> aa3.2 <- asset.allocator(aret3, avar3, risk.aver=.01,
+   max.weight=.45)
> round(aa3.2$new.portfolio * 100, 1)
  equities    bonds commodities
    45.0      10.8      44.2

```

Equities have hit the constraint of 45% in this example.

You can also include an existing allocation. This will be of most interest when trading costs (see Chapter 6) are included. An example (without costs) is:

```

> e60b40 <- c(equities=.6, bonds=.4)
> aa3.3 <- asset.allocator(aret3, avar3, risk.aver=.01,
+   max.weight=.45, exist=e60b40)
> round(aa3.3$new.portfolio * 100, 1)
  equities    bonds commodities
    45.0      10.8      44.2
> round(aa3.3$trade * 100, 1)
  equities    bonds commodities
   -15.0     -29.2      44.2

```

The optimal portfolio remains the same in this case (because there are no trading costs), and the `trade` component of the object returned by `asset.allocator` shows the changes from the existing to the optimal portfolio.

Zero Variance Cash

While some optimizers don't work when the variance is not of full rank—for instance when there is an asset with zero variance, `asset.allocator` (and `portfolio.optimizer`) function perfectly well in such cases. In technical terms this variance matrix is positive semidefinite—it is not positive definite. Here is an expansion of the last example to include cash, which is assumed to have zero variance. (Financially this implies that we have a fixed time horizon with guaranteed interest throughout the time period.)

As before, we create some data to use:

```

> avar4 <- rbind(cbind(avar3, cash=0), cash=0)
> avar4
  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
> aret4 <- c(aret3, cash=3)
> aret4
  equities    bonds commodities    cash
      8         4         5         3

```

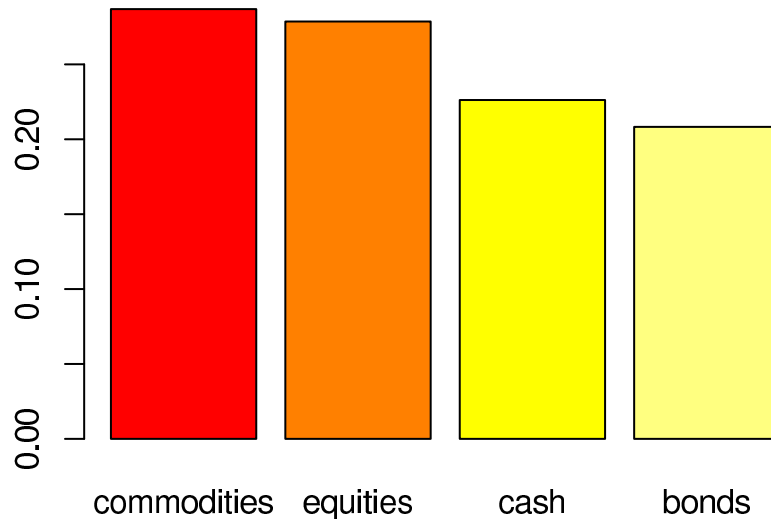
Now we are ready to optimize the allocation:

```

> aa4.1 <- asset.allocator(aret4, avar4, risk.aver=.02)

```

Figure 5.2: Four variable asset allocation.



```
> round(aa4.1$new.portfolio * 100, 1)
  equities    bonds commodities    cash
    27.9     20.8     28.7     22.6
```

Figure 5.2 produces a visual representation of this using the command:

```
> barplot(rev(sort(aa4.1$new.portfolio)))
```

5.3 Asset Allocation with a Benchmark

While asset allocation is seldom performed relative to a benchmark in the traditional sense, it is common for there to be a set of liabilities—the liabilities can be thought of as a benchmark. In practice it seems to be too often the case that the liabilities are ignored in asset allocation.

The `asset.allocator` function does not have the facility of a benchmark, but that is not much of a limitation. Section 5.4 on page 53 explains how to use `portfolio.optimizer` for asset allocation. The current section tricks `asset.allocator` into having a benchmark.

You can modify the variance matrix so that it is relative to the benchmark. Suppose the benchmark is 60% equities and 40% bonds, and we are starting with the `avar3` variance matrix as created on page 49. Commands to modify the variance matrix would be:

```
> e60b40 <- c(equities=.6, bonds=.4)
> avar3b <- var.add.benchmark(avar3, e60b40, "e60b40")
```

```

> avar3b
      equities bonds commodities e60b40
equities  400.0  32.0      24.00 252.80
bonds     32.0  64.0      9.60  44.80
commodities 24.0  9.6     144.00 18.24
e60b40    252.8 44.8      18.24 169.60
> avar3r <- var.relative.benchmark(avar3b, "e60b40")
> avar3r
      equities  bonds commodities
equities  64.00 -96.00      -77.44
bonds    -96.00 144.00      116.16
commodities -77.44 116.16      277.12

```

The process is first to create a variance matrix that includes the benchmark (`avar3b`), and then to transform that matrix into one that is relative to the benchmark (`avar3r`). Note that `avar3r` only has rank 2, so some optimizers will not be able to use this matrix (most optimizers will adjust it slightly).

Once the matrix is built, we can proceed with using it:

```

> aret3r <- aret3 - (8 * .6 + 4 * .4)
> ef.r1 <- efficient.frontier(aret3r, avar3r)
> ef.r1
$values
      Volatility Expected Return Risk Aversion
[1,]  8.0000000    1.600000e+00    0.0000000
[2,]  7.2594108    1.470323e+00    0.0125000
[3,]  6.4998024    1.325161e+00    0.0156250
[4,]  5.9310085    1.209204e+00    0.0171875
[5,]  5.4367523    1.108436e+00    0.0187500
[6,]  4.0775658    8.313271e-01    0.0250000
[7,]  3.2620523    6.650616e-01    0.0312500
[8,]  2.7183768    5.542180e-01    0.0375000
[9,]  2.0387825    4.156634e-01    0.0500000
[10,] 1.0193921    2.078319e-01    0.1000000
[11,] 0.5096957    1.039159e-01    0.2000000
[12,] 0.0000000    2.670887e-09      Inf

$call
efficient.frontier(expected.return = aret3r,
  variance = avar3r)

attr(,"class")
[1] "efffrontBurSt"

```

The three commands create relative expected returns, compute an efficient frontier, and print the efficient frontier object. From the frontier, we decide that a risk aversion of 0.02 is about right, so we perform that optimization:

```

> aa3r.1 <- asset allocator(aret3r, avar3r, risk.aver=.02)
> round(aa3r.1$new.portfolio * 100, 1)

```

```

equities      bonds commodities
  84.2         8.5         7.3

```

Another approach is to maximize the information ratio relative to the benchmark:

```

> aa3r.2 <- asset.allocator(aret3r, avar3r, objective='info')
> round(aa3r.2$new.portfolio * 100, 1)
  equities      bonds commodities
  88.9         2.4         8.7

```

The Mathematics

One formulation for the weights when there is a benchmark is that the weight for the benchmark is minus one while the weights for the assets are as usual.

An equivalent formulation is to use variances and expected returns for the assets minus the benchmark. Thus the variance matrix that we want to use has element (i, j) that is:

$$\text{Cov}(X_i - B, X_j - B) = \text{Cov}(X_i, X_j) - \text{Cov}(X_i, B) - \text{Cov}(B, X_j) + \text{Cov}(B, B)$$

If the benchmark is a combination of the allowable assets, then you can check to see that the variance you are using is correct with:

```

> aa3r.3 <- asset.allocator(aret3r, avar3r, risk.aver=Inf)
> round(aa3r.3$new.portfolio * 100, 4)
  equities      bonds commodities
    60         40         0
> aa3r.3$var.values
[1] -9.377915e-15

```

The allocation with infinite risk aversion should reproduce the benchmark, and should have zero variance. Note that this does not check the validity of the expected return vector as infinite risk aversion ignores the expected returns (actually, POP changes them to be all zero).

5.4 Asset Allocation via the Portfolio Optimizer

It is possible that you may want to use features of `portfolio.optimizer` that are not in `asset.allocator`. One possible reason is to have a turnover constraint.

If there are several dozen assets, then `portfolio.optimizer` optimizes better than `asset.allocator`. It's not clear just where the line is, but probably on the order of 30 or 40 assets.

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

```

> aa.prices <- rep(1, 4)
> names(aa.prices) <- anam4

```

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:

```
> aa4.alt1 <- portfolio.optimizer(aa.prices, avar4,
+   expected.ret=aret4, objective='mean-var', long.only=TRUE,
+   gross.value=10000+c(-.5, .5), risk.aver=.02)
```

performs the same problem as:

```
> aa4.1 <- asset allocator(aret4, avar4, risk.aver=.02)
```

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

```
> valuation(aa4.alt1)$weight
```

caution

In order to have `portfolio.optimizer` act like `asset allocator`, you need to make sure that `ntrade` is at least as large as the number of assets.

5.5 Simultaneous Multiple Scenario Analysis

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 use the variance and expected returns created in Section 5.2 starting on page 48 as the “medium” scenario. We’ll also have a “crash” scenario and a “prosperous” scenario.

```
> 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.crash <- c(35, 16, 20)
> avol.prosper <- c(15, 5, 7)
> avar.crash <- rbind(cbind(t(acor.crash * avol.crash) *
+   avol.crash, 0), 0)
```

S code note

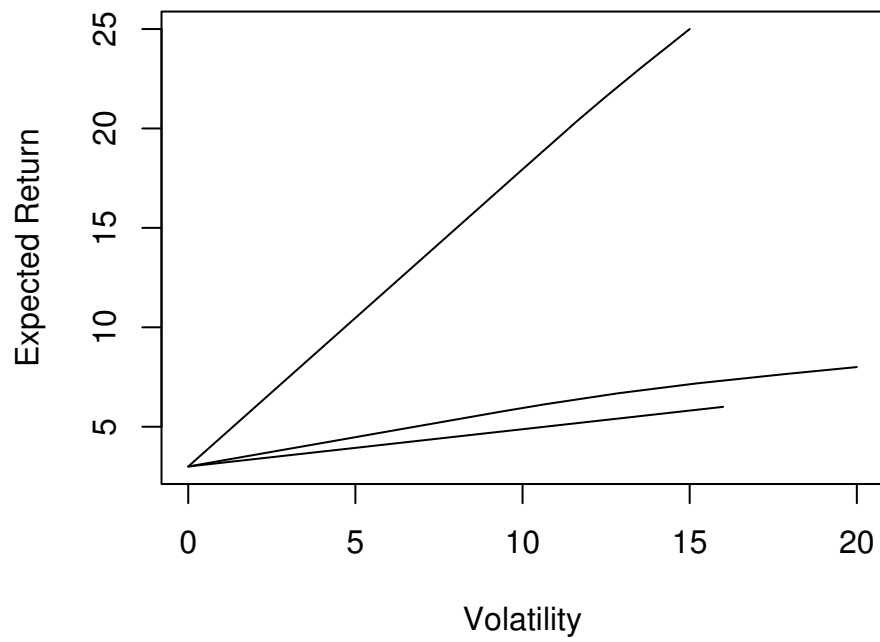
The last command creating `avar.crash` does several things at once. First (in the center of the command) it multiplies the correlation matrix times the volatility vector which multiplies each row of the correlation matrix by the corresponding volatility. Then it transposes the matrix and multiplies again 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.crash) <- list(anam4, anam4)
> dimnames(avar.prosper) <- list(anam4, anam4)
> 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.crash <- c(-20, 6, -9, 3)
> aret.prosper <- c(25, 3.5, 8, 3)
> 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
```

With the data, we can compare the efficient frontiers of the scenarios as they are optimized separately.

```
> ef.medium <- efficient.frontier(aret4, avar4)
> ef.crash <- efficient.frontier(aret.crash, avar.crash)
> ef.prosper <- efficient.frontier(aret.prosper,
+   avar.prosper)
> plot(ef.prosper, xlim=c(0, 20))
> plot(ef.crash, add=T)
> plot(ef.medium, add=T)
```

Figure 5.3: Efficient frontiers under three scenarios.



The plot is given in Figure 5.3.

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 <- array(c(avar.crash, avar4, avar.prosper),
+   c(4, 4, 3))
> dimnames(avar.all) <- list(anam4, anam4, NULL)
```

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. (We also need the order of assets within each matrix to be the same, but that is already true.)

```
> aret.all <- cbind(crash=aret.crash, medium=aret4,
+   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 that we want, we need to do some ground work.

```
> jj <- asset allocator(aret.all, avar.all)
> utab3 <- jj$util.table
> 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.objective   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 of `asset allocator` is to look at 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 `util.table` 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 need to 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.objective   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 `util.table` argument, the `quantile` argument needs to be 1 in order to get a min-max solution:

```
> aa.ms1 <- asset.allocator(aret.all, avar.all,
+   util.tab=utab3, quantile=1)
> round(aa.ms1$new.portfolio * 100, 1)
      equities      bonds commodities      cash
          0.9       34.5           0.0      64.6
> aa.ms1$utility.values
[1] -3.233790 -3.233790 -3.310592
```

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. These two are, unsurprisingly, the crash and medium scenarios. (The optimizer always minimizes, so it is really minimizing the maximum of the negative utilities.)

General Simultaneous Optimization

Min-max optimization protects against the very worst outcome. However, it may really be buying too much insurance. The `asset.allocator` function (and `portfolio.optimizer`) allows less extreme simultaneous optimization as well as min-max. It is just a matter of changing the `quantile` argument to a number that is less than 1 (but only numbers that are at least one-half are sensible):

```
> aa.ms2 <- asset.allocator(aret.all, avar.all,
+   util.tab=utab3, quantile=.7)
> round(aa.ms2$new.portfolio * 100, 1)
      equities      bonds commodities      cash
          0.8       32.7           0.0      66.5
```

In this case the quantity it is minimizing is a weighted average of the two worst utilities.

5.6 Going Farther

Tasks that you might want to undertake:

- To add costs to your asset allocation command, see Chapter 6 on page 61.
- To add constraints, see Chapter 7 on page 69.
- To review common mistakes, see Section 9.1 on page 105.
- To improve your solution or examine the quality of solutions you are getting, go to Chapter 10 on page 113.
- To export your solution to a file, see Section 2.3 on page 23.
- To perform asset allocation with multiple variances, go to Section 12.2 on page 123.

Chapter 6

Trading Costs

This chapter covers trading costs. Costs are the trickiest part of optimization, but are extremely important. The basics of optimization are covered in Chapter 3 for long-only portfolios, Chapter 4 for long-short portfolios and Chapter 5 for asset allocation.

6.1 Background

If there were any justification for the attitude that optimization is hard, it would 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 (see Section 6.5), and the cost of imperfectly known expected returns (see Section 11.5). 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, either because of increased liquidity risk or because of the implicit cost of needing to close the position again.

6.2 Specifying Costs

The specification of costs is very flexible in functions `portfolio.optimizer` and `asset allocator` while simple costs can easily be given. To allow for asymmetry in costs, there are four cost arguments in `portfolio.optimizer`:

- `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.

Since there are only long positions in asset allocation, there are just two cost arguments to `asset allocator`:

- `buy.cost`
- `sell.cost`

If there is no difference between buying and selling, and between long and short positions, then give only the `long.buy.cost` argument in the case of portfolio optimization or `buy.cost` if you are doing asset allocation.

caution

If you give one of the other cost arguments (not `long.buy.cost` and not `buy.cost`) but no others, then the named situation will be the only case where there is cost.

For portfolio optimization costs are given in currency units per (some measure of) the number of asset units. 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.

Costs are relative to the weights in `asset allocator`. In asset allocation everything is considered a buy if no existing portfolio is given.

Linear Costs

Costs, when they are linear, are given to `portfolio.optimizer` in currency units per unit of asset (the same as the `prices` argument).

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 <- portfolio.optimizer(prices, varian, ...,
+   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 <- portfolio.optimizer(prices, varian, ...,
+   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.

Polynomial Costs

Actual costs are unlikely to be linear. If you have a better model than linear for the costs, then you can at least approximate your model of costs. The optimizers take polynomial costs of arbitrary order. Polynomial costs are given by a matrix. The order of the polynomial is one less than the number of columns in the matrix. The first column contains the intercept, the second column has the linear coefficients, the third column has the quadratic coefficients, etc.

caution

Do not forget the intercept.

```
> WRONG <- cbind(linear.coefs, quad.coefs)
> okay.costs <- cbind(0, linear.coefs, quad.coefs)
```

Because it is easy to forget the intercept, a warning is issued if there are non-zero values in any intercept. If you really do have non-zero elements in cost intercepts and you don't want to see the warning about it, then you can set the `do.warn` argument to `c(cost.intercept.nonzero = FALSE)`. Only the first part of the name needs to be given—enough needs to be given so that it is unambiguous relative to the menu of names that can be given `do.warn`.

Consider this example matrix given as the costs:

```
> cost.poly
      [,1] [,2] [,3] [,4]
ABC 0.00 0.01 0.02 0.00
DEF 0.03 0.02 -0.01 0.01
```

If ABC trades 30 lots and DEF trades -25 lots, then the cost for ABC will be:

$$0 + .01(30) + .02(30^2) = 18.3$$

The cost for DEF will be:

$$.03 + .02(25) - .01(25^2) + .01(25^3) = 150.53$$

Note that the matrix needs to have enough columns to encompass the highest order polynomial—lower order polynomials will have zeros for their higher order coefficients.

caution

You should make sure that each polynomial that you give makes sense throughout the allowable trading range of the asset. Negative costs, for instance, are not very reasonable. (Though there may be special circumstances—such as crystallizing a loss for tax purposes—where negative costs are appropriate.)

If your costs are just intercepts—there is a fixed cost for trading any of an asset, then you need to give a two-column matrix with the second column containing all zeros:

```
> intercept.only <- cbind(intercepts, 0)
```

This is most sensible if the intercepts are not all equal—otherwise it is almost the same as constraining the number of assets to be traded.

note

Non-zero intercepts for polynomial costs are not allowed in `asset allocator`, but you still need to give the intercept column (so that it is consistent with `portfolio optimizer`). In portfolio optimization trading is discrete, but in asset allocation it is impossible to tell between no trading in an asset and trading an infinitesimal amount.

If the portfolio is long-short, then there can be more than one type of trading cost for a single asset. For example, an asset that is long in the existing portfolio can be sold heavily enough that it becomes short. There is then the issue of whether or not both intercepts should be counted in the trade. The `doubleconst` control parameter determines which is done. If `doubleconst` is `FALSE` (the default), then only the first intercept is used. If `doubleconst` is `TRUE`, then both intercepts are counted.

Other Nonlinear Costs

While the ability to have polynomial costs is quite flexible, it is probably not flexible enough. 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
> portfolio.optimizer( ..., long.buy.cost=cost.poly)
> portfolio.optimizer( ..., long.buy.cost=cost.poly,
+   cost.par=0:3)
```

These two calls to `portfolio.optimizer` are equivalent (though there would be a slight computational efficiency advantage to the first one).

A more telling use of `cost.par` is:

```
> portfolio.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 coefficient 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.

6.3 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 following this model might look like:

```
> cost.mat <- cbind(commission + spread[names(prices)] / 2,
+   volatility[names(prices)] / sqrt(ave.daily.volume[
+   names(prices)]))
>
> portfolio.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.

[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.

6.4 On Scaling Costs and Expected Returns

The proper scaling of the costs relative to the expected returns (and variance) 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 cost of performing the trade. 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.

Let's revisit the example from Section 6.3:

```
> 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)]
>
> portfolio.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.

See page 120 for an argument of why the costs should be increased from this “rational” value.

There is also the scaling of costs in another sense. The default behavior of `portfolio.optimizer` is to sum the costs from all of the trading and then divide 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.

6.5 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. POP 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.

6.6 Trade-Dependent Costs

You may have a market impact model where the cost for one asset is affected by the amount of trading in other assets. So you have the situation where the optimal trade depends on the costs, but the costs depend on the trade. Two possible approaches to this problem are:

- Perform a number of optimizations using “neutral” costs; evaluate each trade using the impact model; pick the trade that performs best with respect to the utility combined with the improved estimate of costs.

- Perform an optimization using “neutral” costs; iteratively change the costs to reflect the optimal trade.

The first approach is a case where we want the optimization not to be perfect—leaving some variability in the solutions is beneficial (so you would want to set the controls to perform a less rigorous optimization—see Section 10.2). This strategy obviously does not guarantee that you approach the true optimum since the optimizer never sees the true costs of the trade.

The second method, while intuitively appealing, has some practical hurdles. The costs are likely to change to make the current trade look relatively expensive, in which case the optimizer will make the next trade look different from the current trade. It is envisionable that the trades would wander around and eventually come back to where they started. Hence you would want to bound the trade in some fashion to be similar to the current trade. Deciding on the means of bounding and how fast to decrease the size of the bounds seem like non-trivial problems. One tool to bound the trade would be linear constraints on the trade.

6.7 Going Farther

Additional steps might be:

- To look for trouble spots with Chapter 9 on page 105
- To try to get a better understanding of optimization with Chapter 11 on page 117
- To constrain costs (most likely in `random.portfolio`), see Section 7.11

Chapter 7

Constraints

This chapter covers the constraints that are possible in the optimizations. The basics of optimization are covered in Chapter 3 for long-only portfolios, Chapter 4 for long-short portfolios and Chapter 5 for asset allocation.

7.1 Summary of All Constraints

The inherent constraints in asset allocation are that the weights are all non-negative and they sum to 1. Table 7.1 shows the additional constraints that are possible using `asset allocator` and `efficient.frontier`.

The `portfolio.optimizer` function provides several more constraints—Table 7.2 lists the possible constraints along with the arguments used to achieve them. The use of the arguments that control the monetary value of the portfolio is explained on page 27 for long-only portfolios, and on page 35 for long-short portfolios. Trading is also restricted to integer amounts in almost all cases—this is discussed on page 72. `random.portfolio` uses the exact same set of constraints as `portfolio.optimizer` to generate random portfolios.

The `violated` component of the output of the optimizers tells which types of constraints are broken, if any, for the solution found.

Table 7.1: Asset allocation constraints.

Arguments	Constraint
<code>constraint.matrix</code> <code>bounds.constraint</code> <code>trade.constraint</code> <code>abs.constraint</code>	linear constraints on the portfolio and/or the trade
<code>min.weight</code> <code>max.weight</code>	upper and lower bounds on the weight of each asset
<code>sum.weight</code>	upper bound on the sum of a specified number of the largest weights
<code>var.constraint</code>	upper bound on the variance of the asset allocation
<code>alpha.constraint</code>	lower bound on the expected return of the asset allocation

Table 7.2: Portfolio optimization constraints.

Arguments	Constraint
ntrade	number of assets to trade
port.size	number of assets in the portfolio
gross.value net.value long.value short.value long.only	monetary value of the portfolio
trade.value	value of the trade (turnover)
constraint.matrix bounds.constraint trade.constraint abs.constraint	linear constraints on the portfolio and/or the trade
lower.trade upper.trade	lower and upper bounds on the number of asset units to trade
universe.trade bench.trade	restrict assets to be traded
max.weight	maximum weight in the portfolio per asset
sum.weight	maximum of the sum of a specified number of the largest weights
var.constraint	bound on variance of the portfolio
bench.constraint	bound on squared tracking error
alpha.constraint	bound on an expected return of the portfolio
threshold	minimum number of asset units to trade or hold
limit.cost	allowable range of costs
close.number	number of positions to close
forced.trade	trades that must be done (at minimum)

7.2 Integer Constraints

The `portfolio.optimizer` function has several types of integer constraint. In addition to the constraints discussed in this section, threshold constraints are presented in Section 7.8.

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. This has a relatively small default value since the speed of the optimization is somewhat sensitive to the value of `ntrade`—the larger the value, the slower the optimization. If you do not want a limit on the number of assets traded, then set `ntrade` to the size of the universe.

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 7.8) are used as well. Otherwise trading just one unit of an asset counts.

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

Number of Assets in the Portfolio

The `port.size` argument allows you to constrain the number of assets in the constructed portfolio. Give it the lower and upper bound of the number of assets that are allowed in the portfolio. If a single number is given, that will only be the upper bound.

If you are optimizing and you want to have an exact number of assets in the portfolio, it will be better to begin by giving a range for the portfolio size with the upper limit being the size you want, `c(50, 55)` for example. Once that optimization has been done, then you can set `port.size` to be exact (with `c(55, 55)` for example). This is because of the optimization algorithm.

There is no problem with specifying an exact portfolio size when generating random portfolios.

Number of Positions to Close

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.

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

Round Lots

Trading is only done in integer amounts except when closing out an existing position that is non-integral. Thus if the prices are given for lots as opposed to shares, the optimizer does round lotting as an integral part of the optimization.

7.3 Constraining Turnover

An easy alternative to using trading costs is to put a turnover constraint onto the problem. If the trading costs are linear and all equal (for example, 50 basis points of the value traded), then a turnover constraint is almost equivalent. The problem with providing a turnover constraint is that you don't know what the optimal constraint is—the farther from optimal the existing portfolio is, the larger the turnover constraint should be.

When a more studied approach is used for trading costs, turnover is no longer equivalent. However, trading costs and a turnover constraint could both be employed.

The `trade.value` argument to `portfolio.optimizer` constrains turnover. The value of the trade is considered to be the amount of money in both the buys and the sells.

While generally only an upper bound on turnover is desired, a minimum can also be given. Do this by giving a length two vector to `trade.value`. While seldom of interest for optimization, this is likely to be useful for generating random portfolios.

7.4 Lower and Upper Trading Bounds

There are no trading bounds for asset allocation—this section only applies to portfolio optimization and random portfolio generation.

One use of these arguments is to present the optimizer with a selection of assets to buy and a selection of assets to sell as seen on page 38 for long-short portfolios and on page 32 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
> optim.lq1 <- portfolio.optimizer(prices, varian, ...,
+   upper.trade = liquidity, lower.trade = -liquidity)
```

Here and in subsequent examples the “...” refers to `portfolio.optimizer` arguments that you need—see Chapter 3 or Chapter 4 for what you might want to use.

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 the optimizer will just ignore those. 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 rest of the optimization. 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 optimization is in lots.

Another sort of liquidity constraint is to limit the amount of assets in the portfolio to some value—perhaps n days of average volume. This is slightly more complicated since we want the bounds on the portfolio, but the lower and upper constraints are on the trade.

With the vector of maximum holdings, you can create the upper bound on the trade relative to the current holdings:

```
> max.hold <- 10 * ave.daily.volume
> jj.cn1 <- intersect(names(max.hold), names(cur.port))
```

S code note

The `intersect` function returns a vector of the unique values that are common to both of its arguments.

```
> max.trade <- max.hold
> max.trade[jj.cn1] <- max.hold[jj.cn1] - cur.port[jj.cn1]
```

The above commands pick out the assets that are common in the maximum holding vector and the current portfolio and modify these to create the maximum to trade. To get both types of liquidity constraint, the two need to be combined using the `pmin` function:

```
> up.lim <- pmin(max.trade, liquidity)
> optim.lq2 <- portfolio.optimizer(prices, varian, ...,
+   exist=cur.port, upper.trade = up.lim,
+   lower.trade = -liquidity)
```

The optimization above assumes a long-only portfolio.

caution

Do not use `min` when you want to use `pmin`. The `min` function returns a single number which is the minimum of all of the numbers in all of the vectors given it. The `pmin` function (as in parallel minimum) returns a vector which is the minimum element by element.

Another way of getting a parallel minimization wrong can be illustrated by looking at the case of a long-short portfolio. The call to `pmin` above was very simple because both arguments are directly derived from the same vector (`ave.daily.volume`) so they have the same assets in the same order.

For a long-short portfolio we want to limit the size of short positions as well as long positions. The source of the maximum short positions may well be different, so the order and number of assets may be different. To combine them, we need to make sure that we always compare numbers for the same asset.

In the following we assume that `max.short` is a vector of positive numbers giving the maximum to short a set of assets.

```
> low.lim <- -liquidity
> jj.cn2 <- intersect(names(cur.port), names(max.short))
> min.sh.tr <- -max.short
> min.sh.tr[jj.cn2] <- min.sh.tr[jj.cn2] + cur.port[jj.cn2]
> jj.cn3 <- intersect(names(low.lim), names(min.sh.tr))
> low.lim[jj.cn3] <- pmax(low.lim[jj.cn3], min.sh.tr[jj.cn3])
> optim.lq3 <- portfolio.optimizer(prices, varian, ...,
+   exist=cur.port, upper.trade = up.lim,
+   lower.trade = low.lim)
```

The `min.sh.tr` vector gives the limit for the portfolio holding; `low.lim` starts off as the limit for trading, and finally (after the call to `pmax`) holds the combination of limits.

7.5 Simple Weight Constraints

Asset Allocation

The `min.weight` and `max.weight` arguments to `asset.allocator` restrict the minimum and maximum weights in the allocation for each asset. These may be given as a single number, in which case all assets have the same limit. Alternatively, a named vector may be given—the named assets are restricted by the value, unnamed assets remain unrestricted.

Portfolio Optimization

There is no `min.weight` argument for portfolio optimization. Somewhat analogous to this are lower and upper bounds on trading (Section 7.4) or threshold constraints (Section 7.8).

The `max.weight` argument is very similar to the asset allocation argument, but there is a difference in how it treats unnamed assets. Typical use of this argument is:

```
> optim.wc1 <- portfolio.optimizer(prices, varian, ...,
+   max.weight=0.05)
```

This command 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.

In some cases the desired limits are more complex than this. If you want to force a few assets to have smaller weights than the rest, then give a named vector of the smaller bounds. The remaining assets will be bounded by the maximum of the given bounds so include one of the non-special assets as well.

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.

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.

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 optimized 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`. To just constrain the largest weight, the `sum.weight` argument (see Section 7.6) would be similar to:

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

Alternatively, if there are additional sums of weights to be constrained, it might be:

```
c("1"=.1, "5"=.4, "10"=.7)
```

7.6 Sums of Largest Weights

Another type of constraint is that the largest n weights should sum to no more than some limit—this type of limit is handled by the `sum.weight` argument. This argument exists for both portfolio optimization and asset allocation.

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%, your command would look like:

```
> optim.wc2 <- portfolio.optimizer(prices, varian, ...,
+   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 doing the same thing would be:

```
> sumw.arg <- c(.4, .7)
> names(sumw.arg) <- c(5, 10)
> optim.wc2 <- portfolio.optimizer(prices, varian, ...,
+   sum.weight=sumw.arg)
```

7.7 Linear Constraints

Well-reasoned constraints on such things as sectors or countries are a useful part of optimization. However, too tight of constraints leaves the optimizer with little to do but try to satisfy the constraints—in such cases there is little optimality. Random portfolios (Chapter 8) are a valuable tool to examine constraints.

Though at heart they are the same, POP 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. (Most other optimizers will expand such constraints into a matrix of zeros and ones—you could do this also, but the method used is more compact as well as easier for the user.)

note

The `asset allocator`, `portfolio.optimizer` and `random.portfolio` functions take linear constraints precisely the same. The only difference is that constraints are computed on weights in asset allocation, but on money in the other two functions. In summary:

- `asset allocator` and `efficient.frontier`: constraints in weights
 - `portfolio.optimizer` and `random.portfolio`: constraints in monetary units
-

Building Constraints

Constraints involve at least two arguments. The `constraint.matrix` argument gives the numbers or categories for each asset for each constraint. The other mandatory argument is `bounds.constraint` which provides the lower and upper bound for each (sub)constraint. Because the optimizers 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 `constraint.matrix` argument of the optimizers. The vector or matrix can be either character or numeric. A data frame is required if you have a mixture of numeric and categorical constraints.

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] "matrix" "bounds"
```

The result of `build.constraint` is a list with two components. The `matrix` component is suitable to give as the `constraint.matrix` argument, the `bounds` component is the template for an object to give as `bounds.constraint`. The constraint matrix must have column names when it is given to the optimizer, 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$matrix[1:3,]
  country sector
ABC "Spain" "media"
DEF "France" "retail"
GHI "Italy" "energy"
> cons.obj$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.

```
> bounds.cs <- cons.obj$bounds
> bounds.cs[1:2, ] <- c(1e5, 2e5, 4e5, 5e5)
> bounds.cs[3, 2] <- 2e5
> bounds.cs[5,] <- c(3e5, 6e5)
> bounds.cs
      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

```

For portfolio optimization the bounds are in currency units—if you think in terms of weights, then multiply the weights by the gross value that you want. There is no need for any particular pattern of finite bounds. In this example, the only sector we are bounding is media. We can now proceed to an optimization:

```

> opti1 <- portfolio.optimizer(prices, varian, ...,
+   constraint.mat=cons.obj$matrix, bounds=bounds.cs)

```

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.cs)

```

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.cs` and will have infinite bounds for the industries.

The bounds that are given to the optimizers 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 in the problem.

Only categorical constraints have been discussed so far. To see an example of numeric constraints, and a mixture of numeric and categorical constraints, see Section 4.6 on pairs trading (though this could be thought of as categorical as well).

Constraints on Trades and/or Absolute Values

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

By default, constraints use the value (or the weight in asset allocation) for each asset. If you want the constraints on the absolute value, then set the `abs.constraint` argument to be `TRUE`. In a long-short portfolio you may want to limit the net exposure to a sector, but you may also want to limit the gross amount in the sector. The `abs.constraint` argument is also replicated to have length equal to the number of columns in the constraint matrix.

Absolute and non-absolute constraints are the same for asset allocation or long-only portfolios unless the constraint is on the trade. As with trade constraints, there can be a mixture of absolute and non-absolute constraints.

There is a total of four types of linear constraint:

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

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))
> bound.4c <- con.4c$bound
>
> # impose desired bounds by modifying bound.4c
>
> opti2 <- portfolio.optimizer(prices, varian, ...,
+   constraint.mat=con.4c$matrix, bounds=bound.4c,
+   trade.con=c(T,T,F,F), abs.con=c(F,T))
> opti2[c("trade.constraint", "abs.constraint")]
$trade.constraint
  c.tn c.ta c.pn c.pa
  TRUE TRUE FALSE FALSE

$abs.constraint
  c.tn c.ta c.pn c.pa
  FALSE TRUE FALSE TRUE
```

Looking at the Effect of the Constraints

The `violation.constraints` component of the object returned by the optimizers tells you the status of the linear constraints. (Using `summary` can give you information on some of the other constraints as well.) Here is an example:

```
> opti2$viol
      lower upper  value 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 is a matrix where each row contains information on each 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 achieved value is below the lower bound, and a positive violation means that the achieved value is above the upper bound. The next to last column gives proximity of the achieved value to the nearest bound. The same convention in terms of sign is used—a negative number means the achieved is closer to the lower bound.

In this example the sizes of the violations are relatively trivial. However, they are not trivial in terms of their affect on the objective—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.

Random portfolios (see Chapter 8) provide a means of testing how tight the bounds on constraints should be.

Evaluating Un-imposed Constraints

It is possible to look at the value of constraints that were not imposed on a portfolio. Give `violation.constraints` the portfolio and the matrix of constraints that are of interest:

```
> violation.constraints(opt.ir2, cbind(country=countryvec))
      lower upper  value nearest violation
country : France -Inf  Inf  20106  -Inf      NA
country : Italy  -Inf  Inf  156728  -Inf      NA
country : Spain -Inf  Inf -157020  -Inf      NA
```

You may also use the `abs.constraint` argument to specify if the constraints are to be absolute or not, and the `trade.constraint` argument to state if constraints are for the trade or the portfolio:

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

7.8 Threshold Constraints

The `threshold` argument controls two types of threshold constraints—trade thresholds and portfolio thresholds.

Trade Thresholds

Trade threshold constraints force the optimizer to trade at least a certain amount of an asset if it is traded at all.

The command:

```
> op.th1 <- portfolio.optimizer(prices, varian, long.only=T,
+ gross=1e6, threshold=c(ABC=4, DEF=23))
```

requests 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 7.10.

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:

```
> op.th1b <- portfolio.optimizer(prices, varian, long.only=T,
+ gross=1e6, threshold=rbind(ABC=c(-4,4), DEF=c(-23,23)))
```

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

```
> op.th2 <- portfolio.optimizer(prices, varian, long.only=T,
+ gross=1e6, 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 optimized 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
> op.th2 <- portfolio.optimizer(prices, varian, ...,
+   threshold = thresh.m1)

```

This states that 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.

7.9 Constraints on Variances, Tracking Errors and Expected Returns

Constraints can be placed on the variance (or several variances if more than one variance is used). Likewise constraints can be placed on tracking errors and on expected returns.

Variance Constraints

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.

Perhaps its next most common usage is a two column matrix such as:

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

that gives a range of values that the variance is allowed to be in. This can be done in `portfolio.optimizer` and `random.portfolio`, but `asset allocator` does not accept this use.

advanced

This argument can be a vector (with one or more values) with or without names, or it can be a one- or two-column matrix. A plain vector is equivalent to a one-column matrix. The row names, if they exist, should be character representations of zero-based indices of the variance. So:

```
var.constraint = c('0'=1.2, '1'=1.4)
```

means that the first variance is constrained to be no more than 1.2 and the second variance is constrained to be no more than 1.4.

A very similar command means something quite different:

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

means that the (first) variance is constrained to be at most 1.4 but no less than 1.2. That is, 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 could potentially be 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 three variances are given—that is, the `variance` argument is a three-dimensional array that is number of assets by number of assets by 3. Then the command:

```
var.constraint = varcon.m1
```

would say that the first variance (strictly speaking the first variance-benchmark combination) is restricted to be between 1.2 and 1.4, the second variance is restricted to be between 1.3 and 1.5, and the third variance 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 is to be between 1.2 and 1.4, the first variance is restricted to be between 1.3 and 1.5, and the second variance is unconstrained.

Benchmark (Tracking Error) Constraints

This subsection applies only to `portfolio.optimizer` and `random.portfolio`. The functionality is not available for `asset allocator`.

The `bench.constraint` argument is used to constrain (squared) tracking errors. Examples of its most common use are on page 30 (simple use) and page 31 (multiple benchmarks).

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 tracking error to `spx` to between 2% and 4%. This is most likely of interest when generating random portfolios.

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

Alpha (Expected Return) Constraints

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 estimated expected return.

In `portfolio.optimizer` and `random.portfolio` an upper bound as well as a lower bound can be specified by giving a two-column matrix. For instance:

```
alpha.constraint = rbind(c(1.1, 2.3))
```

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

Advanced use of `alpha.constraint` is analogous to the advanced use of `var.constraint` (page 82) except that a one-column matrix is a lower bound rather than an upper bound, and indices refer to columns of `expected.return` rather than to variance-benchmark combinations.

7.10 Forced Trades

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 lots of ABC and buy at least 15 lots of DEF. If you wanted to buy exactly 15 lots 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 75).

7.11 Cost Constraints

While unlikely to be of use with optimization, it is possible to put upper and lower constraints on the cost. This can be very useful for mimicking actual trading with random portfolios. This can not be done in `asset allocator`.

The argument to constrain costs is `limit.cost`. If it is given, then it must be a length two vector giving the allowable range of costs.

7.12 Constraint Penalties and Soft Constraints

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, and the constraint on the number of assets traded is always obeyed.

Tools are available in `portfolio.optimizer` to allow you to easily 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:

```
> sc.opt1 <- portfolio.optimizer(prices, varian, alphas,
+   gross.value=1.4e6, long.only=TRUE, existing=cur.port,
+   constraint.mat=conmat, bounds=boundmat,
+   var.constraint=370, sum.weight=c('5'=.15),
+   max.weight=.05, ntrade=30)
> sc.opt1$results
objective      cost  penalty
370083.6       0.0  370083.8
```

Since the penalty element of the `results` component of the output is not zero, not all of the constraints are satisfied. The `violated` component is a character vector of the constraints that are broken:

```
> sc.opt1$violated
[1] "sum of weights" "linear"          "gross value"
```

But it is the `constraint.violations` component that lists by how much each constraint is broken:

```
> sc.opt1$constraint.violations
      Country      gross      net
216.8500    0.6000    0.0000
      long      short      tradeval
0.0000    0.0000    0.0000
port.size      variance 0      sum.weight
0.0000    0.0000    152.6338
      cost      close      forced.trade
0.0000    0.0000    0.0000
min ntrade      trade threshold portfolio threshold
0.0000    0.0000    0.0000
```

From this we can infer that there is only one column in the constraint matrix and it is called “Country”. In addition to the linear constraint, there is an entry for the constraint on the variance. All of the other items are always in the vector even if there are no such constraints in the problem.

The penalty that is imposed (the value in the results component) is the sum of the constraint violations times the penalties.

```
> sum(sc.opt1$constraint.violations *
+     sc.opt1$penalty.constraint)
[1] 370083.8
```

The default value of `penalty.constraint` 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.

```
> sc.opt2 <- portfolio.optimizer(prices, varian, alphas,
+   gross.value=1.4e6, long.only=TRUE, existing=cur.port,
+   constraint.mat=conmat, bounds=boundmat,
+   var.constraint=370, sum.weight=c('5'=.15),
+   max.weight=.05, ntrade=30,
+   penalty.con=c(Country=.1, 'variance 0'=20))
```

7.13 Quadratic Constraints

The ability to accept more than one variance means that `portfolio.optimizer` and `random.portfolio` allow quadratic constraints on the portfolio. There are three steps when creating quadratic constraints: the constraint matrices need to be added to the variance, the constraint bounds need to be specified, and the optimizer needs to be told not to include the constraints in the utility.

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.

`c` 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.

The next step is to impose the constraint bounds. This will be done with the `var.constraint` argument as discussed in Section 7.9. 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 as opposed to linear constraints where the bounds are in terms of money.

The final step is to tell the optimizer that the final two “variances” are not to be used in the utility. We do this by first performing a dummy run, getting some output to change, and then using that changed object as input to the actual optimization.

```
> qc.temp <- portfolio.optimizer(prices, varian3,
+   funeval.max=2, ...)
> qc.utiltab <- qc.temp$util.table
> 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
> qc.opt <- portfolio.optimizer(prices, varian3,
+   util.table = qc.utiltab, var.con = ...)
```

We have merely taken the default utility table, changed it slightly, and then use that in the actual optimization. The change is to send the second and third variances to destination -1, that is, not to use them in the utility. (Only destinations 0 and greater count in the utility.) See Section 13.4 for more on this.

7.14 Going Farther

Additional tasks might include:

- Generate random portfolios in order to evaluate if your constraints are too loose or too tight. This is explained in Chapter 8 on page 89.
- Consult Chapter 9 on page 105 to check for common mistakes and other practicalities.

Chapter 8

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 the utility or costs.

8.1 Uses

There are a number of uses for random portfolios. These include:

- Measuring fund manager performance.
- Evaluating the effect of constraints.
- Testing the efficacy of the expected returns prediction process.
- Providing the basis of an investment mandate.
- Examining the opportunity available to a strategy.
- Assessing the quality of a risk model.
- Portfolio construction process attribution.
- Benchmarking mandate breaches.
- Calculating a bid on a portfolio whose composition you do not know, but which has some known characteristics.
- Optimizing a non-standard utility.

Some of these are suggested in [Dawson and Young, 2003]. They are looked at in more depth below. This list is undoubtedly incomplete—it is the result of a few people with little experience thinking a little bit about the possibilities. The statistical bootstrap [Efron, 1979] has gone from a nice idea to a central tool of statistical analysis (and a major topic of theoretical statistical research). Random portfolios are quite analogous to bootstrapping—they are likely to become a key tool for fund management at some point.

Fund Manager Performance

The “benchmark” is a set of random portfolios that mimic the constraints that the fund manager uses (but with no constraint on expected returns or utility). The fund manager should outperform many of the random portfolios. This is a more fair comparison than other methods (such as against the manager’s peers) as the random portfolios can be made to have almost exactly the same constraints as the fund manager. The fraction of random portfolios with performance that is as good as or better than the fund manager is the p-value for the statistical hypothesis test that the manager has skill (for that period).

Freedom is one of the great benefits of this. Without random portfolios fund managers need to be exceedingly close to a benchmark in order to have much hope of proving their superiority over the benchmark. This is bad for the fund manager since the closer they are to the benchmark, the more logical it is for investors to invest passively in the benchmark. It is bad for the investors because they have less opportunity to buy funds that are superior to the benchmark and are diversified from it. Random portfolios allow fund managers to be assessed for their skill in the arenas where they believe they can add most value.

More details are in [Burns, 2004].

The Effect of Constraints

One example of this is to explore the amount of turnover that is required to stay within constraints—such as tracking error and sector constraints—that are imposed on a fund.

[Burns, 2003a] uses random portfolios to explore how tightly market neutral funds should constrain their beta. It also explores the market neutrality of constraining the net value of the fund to be close to zero (without a constraint on beta such funds are not very neutral at all).

If constraints are too tight, then solutions will be constrained away from optimality. If constraints are too loose, then they will not be performing their intended function. Random portfolios can tell you how useful your constraints are, and suggest how tight they should be.

The Efficacy of Returns Prediction

Two sets of random portfolios are generated. Both contain the constraints—such as tracking error and maximum weight—that are a part of the fund management process. However, one of them will have the additional constraint that the expected return must be above some value. The performance of the portfolios in an out-of-sample period can be computed. The difference in returns for the two groups can then be compared statistically.

Section 8.3 provides an example of another approach to doing this. Other schemes may be used as well.

The Basis of an Investment Mandate

A mandate can be created that specifies the constraints that should be obeyed by the fund manager. Random portfolios can be used to measure how well the fund manager performs. Performance fees can be based on random portfolios if desired. [Burns, 2004] provides more specifics.

Opportunity Analysis

In general the potential that a strategy has depends on the cross-sectional behavior of the appropriate universe of assets. If all of the assets act precisely the same, then there would be no possibility of outperforming. Random portfolios can be used to explore how large opportunities are, and to document their changes over time.

The Quality of a Risk Model

The realized volatility or tracking error of random portfolios can be compared with what a risk model would have predicted for the portfolios. Various constraints should be imposed on the random portfolios that are generated so that they are realistic. In particular, a constraint on market capitalization is a good idea as long-only random portfolios without this constraint are likely to have a bias towards small caps.

Portfolio Construction Process Attribution

To the extent that the portfolio construction process can be broken into pieces, random portfolios can be generated to imitate each piece of the process. The value of each piece can then be assessed. See [Bridgeland, 2001].

Benchmarking Mandate Breaches

If a fund has breached some aspect of a mandate—such as a limit on tracking error, then the generation of random portfolios can help resolve whether such an event was essentially unavoidable or can be considered to be carelessness.

Bidding on Portfolios

An example is shown in Section 8.4.

Optimizing a Non-standard Utility

If you want to optimize on a utility that is beyond the portfolio optimizers at your disposal, then you can use random portfolios as the ingredient for an optimization using the totally random algorithm. This is not a computationally efficient algorithm, but is almost surely faster than creating a new optimizer. See Section 13.5 for slightly more on this algorithm.

The process is to first generate random portfolios that satisfy the constraints that you would like. Then evaluate your utility for each of the portfolios that have been created. The portfolio with the best utility is your “optimal” portfolio.

8.2 The Command

The generation of random portfolios is done virtually the same as the optimization of portfolios. Any `portfolio.optimizer` arguments can be given. However, the first argument to `random.portfolio` is the number of random

portfolios that you want to generate. There is also the `out.trade` argument which controls whether it is the random portfolio (the default) or the trade which is output.

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

```
> randport1 <- random.portfolio(100, prices, varian,
+   long.only=T, bench.constr = c(spx=.04^2/252),
+   constraint.mat=cntrysect.conmat, ntrade=55,
+   bounds.con=cntrysect.bounds, gross.val=1e6)
```

This just adds 100 as the first argument to a set of arguments that would be given to `portfolio.optimizer`.

You can see how many assets are in the portfolios, and the number of times each asset is in a portfolio with `summary`:

```
> summary(randport1)
```

Some times 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`:

```
> randtrade1 <- random.portfolio(100, prices, varian,
+   long.only=T, bench.constr = c(spx=.04^2/252),
+   constraint.mat=cntrysect.conmat, ntrade=55,
+   bounds.con=cntrysect.bounds, gross.val=1e6,
+   out.trade=T)
```

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.)

One reason to generate a list of the random trades, is that the trades can be used as the `start.sol` argument with `funeval` set to 0 in order to evaluate such things as expected returns and tracking errors of the random portfolios. Section 10.1 on page 113 has a little more detail on this process.

```
> rand.alpha1 <- numeric(length(randtrade1))
```

S code note

The `numeric` function creates a vector of numbers with length equal to its first argument. It is more efficient in S to create an object the correct length and then change each element than to build up an object by adding each new result to the old results.

```

> for(i in 1:length(randtrade1)) {
+   rand.alpha1[i] <- portfolio.optimizer(prices, varian,
+     long.only=T, bench.constr=c(spx=.04^2/252),
+     constraint.mat=cntrysect.conmat,
+     bounds.con=cntrysect.bounds, gross.val=1e6,
+     funeval=0, start.sol=randtrade1[[i]])$alpha.val
+ }

```

The `rand.alpha1` vector is filled with the expected return of each portfolio that results from the trades in `randtrade1`.

Exporting Random Portfolios

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

```

> deport(randport1)
[1] "randport1.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).

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

```

> deport(valuation(randport1, prices), file="randval1")
[1] "randval1.csv"

```

Specifying the file name is preferred in this case as the default will not be a very good choice.

If you would like a compact form of the file when there is a large universe of assets, you can use the `append` argument and iteratively write each portfolio to the file.

```

> deport.randportBurSt(randport1[1], file="randcompact")
> for(i in 2:length(randport1)) {
+   deport.randportBurSt(randport1[i], file="randcompact",
+     append=TRUE)
+ }

```

We start by first writing just the first portfolio with the default value of `append`, which is `FALSE`. The full name of the `deport` function needs to be used because the result of the subscripting has lost the class attribute. Specifying the file name is a necessity in this case. The remaining portfolios are written by a loop with `append` set to `TRUE`.

Working with Random Portfolios

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. Using the `c` function puts 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 hasn't received 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.

8.3 Example: Exploring Alpha Generation

Evaluating the quality of prediction for a single asset is reasonably easy. However, what really matters is the out-performance of the portfolio. This is significantly harder to appraise. It depends on the combination of individual predictions, on the portfolio constraints, and on the balance between risk and returns.

The example is a long-only portfolio with some linear constraints, the maximum weight constrained, and an upper bound on the variance. Note that the alpha generation in the example is pure fabrication, and the results are unlikely to reflect actual results.

We are supposing that this is a part of a backtest. This optimization uses information available up to the beginning of period 1. The optimization is subsequently evaluated over period 1. The same procedure would be used for the remaining periods.

```
> ag.opt1 <- portfolio.optimizer(prices, varian, alphas,
+   ntrade=35, long.only=T, gross.val=1.6e6,
+   constraint.mat=cs.commat, bounds=cs.bounds * 1.6e6,
+   max.weight=.05, var.constrain=3e-5)
```

Once the optimization is performed, we need the weights of the assets within the optimal portfolio:

```
> ag.opt1wt <- valuation(ag.opt1)$weight
```

With the weights we can calculate the returns of the portfolio over the period of interest:

```
> ag.opt1ret <- drop(retmat.p1[, names(ag.opt1wt)] %*%
+   ag.opt1wt)
```

S code note

The `%*%` operator is matrix multiplication. The `retmat.p1` matrix is assumed to have dates along the rows and assets along the columns. We select the pertinent columns of `retmat.p1`, do the multiplication, then drop the matrixness of the result to make it a vector. The result is a vector over time of the returns of the optimal portfolio.

The next step is to generate a number of random portfolios with the same constraints as the optimal portfolio:

```
> ag.rand1 <- random.portfolio(100, prices, varian,
+   alphas, ntrade=35, long.only=T, gross.val=1.6e6,
+   constraint.mat=cs.conmat, bounds=cs.bounds * 1.6e6,
+   max.weight=.05, var.constrain=3e-5)
```

This command has the same arguments as the optimization command except that the first argument, the number of random portfolios to generate, is added.

We now want to create return series for the random portfolios like we did for the optimal portfolio. We first create a function that computes the weights in a portfolio, and apply that to the list of random portfolios.

```
> fun.wt <- function(x, prices)
+ {
+   valuation(x, prices)$weight
+ }
> ag.rand1wt <- lapply(ag.rand1, fun.wt, prices=prices)
```

S code note

The `lapply` function applies the function given as the second argument to each component of the first argument. The arguments to `lapply` are often just a list and a function. But additional arguments for the function can be given as arguments to `lapply`.

We could have put the definition of the function in the call to `lapply`, there is no need to give the function a name.

S code note

The `valuation` function is generic, meaning that it really just asks another function to do the work. The function that is dispatched depends on the class of an argument (almost always the first argument). There is a method of `valuation` for the random portfolio class, but that is not what is used in `fun.wt`. There `valuation` is being used on individual components, which do not have a class so the default method is used.

Next we write another function that produces a vector of returns for a portfolio given the portfolio weights and a matrix of asset returns. We use this function to get return series for each random portfolio.

```
> fun.ret <- function(wt, retmat)
+ {
+   drop(retmat[, names(wt)] %*% wt)
+ }
> ag.rand1ret <- do.call("cbind", lapply(ag.rand1wt,
+   fun.ret, retmat=retmat.p1))
```

S code note

The use of `do.call` with `cbind` is a shortcut to end up with a matrix of returns where each column represents a different random portfolio.

The command:

```
> do.call("cbind", list(1:4, 5:8))
```

is equivalent to:

```
> cbind(1:4, 5:8)
```

We now make a vector that has just a single number—the return over the whole period—for each of the random portfolios:

```
> ag.rand1retsum <- colSums(log(ag.rand1ret + 1))
```

The log return over a period of time is the sum of log returns within the time period. We have been (implicitly) assuming to this point that we had simple returns since those are what work for summing weights into a portfolio return.

Table 8.1 summarizes the objects that have been created in this example.

We are ready to explore the quality of the optimal portfolio. An easy thing to do is draw a histogram of the returns over the period for the random portfolios, and then indicate where on the graph the optimal portfolio lands.

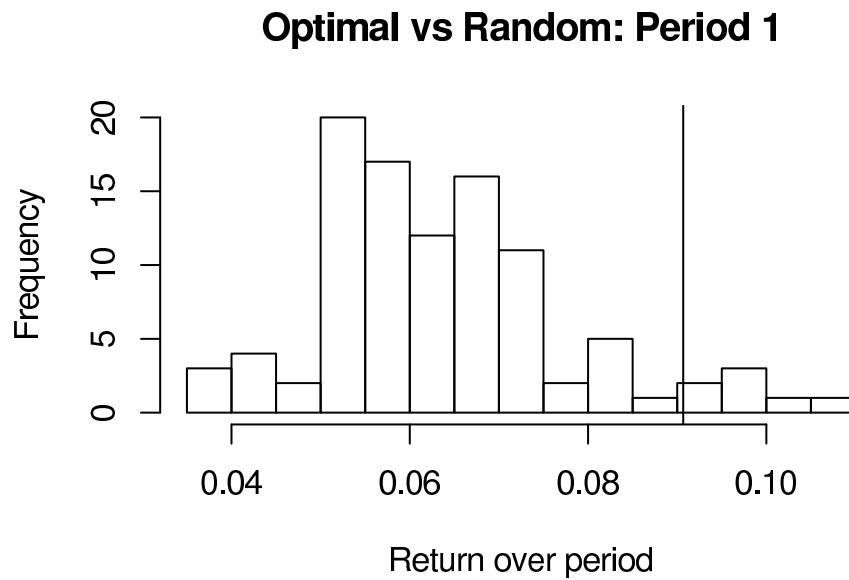
```
> hist(ag.rand1retsum, 20)
> abline(v=sum(log(ag.opt1ret + 1)))
```

While the command above is likely to be done in an interactive session, the following command is what might be used for a more formal plot. This is the command that produced Figure 8.1:

Table 8.1: S objects for alpha generation example.

object	type	description
ag.opt1wt	vector	optimal portfolio weights
ag.opt1ret	vector	returns of optimal portfolio over time
fun.wt	function	produces portfolio weights
ag.rand1wt	list	random portfolio weights
fun.ret	function	produces portfolio returns
ag.rand1ret	matrix	random returns (rows:time; columns: portfolio)
ag.rand1retsum	vector	period returns for random portfolios

Figure 8.1: Optimal portfolio return relative to random returns.



```
> hist(ag.rand1retsum, 20, xlab="Return over period",
+       main="Optimal vs Random: Period 1")
> abline(v=sum(log(ag.opt1ret + 1)))
```

One question of interest is what fraction of the random portfolios had a better return than the optimal portfolio. This is answered by the following command:

```
> mean(sum(log(ag.opt1ret + 1)) < ag.rand1retsum)
[1] 0.07
```

S code note

Though very simple, this is a cryptic command to the uninitiated. We compare the period return for the optimal portfolio with the period returns for the random portfolios. This gives us a vector as long as the number of random portfolios which is `TRUE` if the random portfolio had a better return and `FALSE` otherwise.

We then ask for the mean of this logical vector. Since the mean is a numerical operation, the logical values get changed into numbers—`TRUE` into 1 and `FALSE` into 0. So the mean gives us the number of `TRUE` values divided by the total number of values.

We are supposing that this exercise of comparing optimal portfolios with random ones is carried out for a number of periods. We could then plot the fraction of random portfolio outperformance over time, which might look like Figure 8.2.

```
> plot(ag.outper * 100, xlab="Period",
+       ylab="Probability random is better (%)", type="l")
```

8.4 Example: Bidding on an Unknown Portfolio

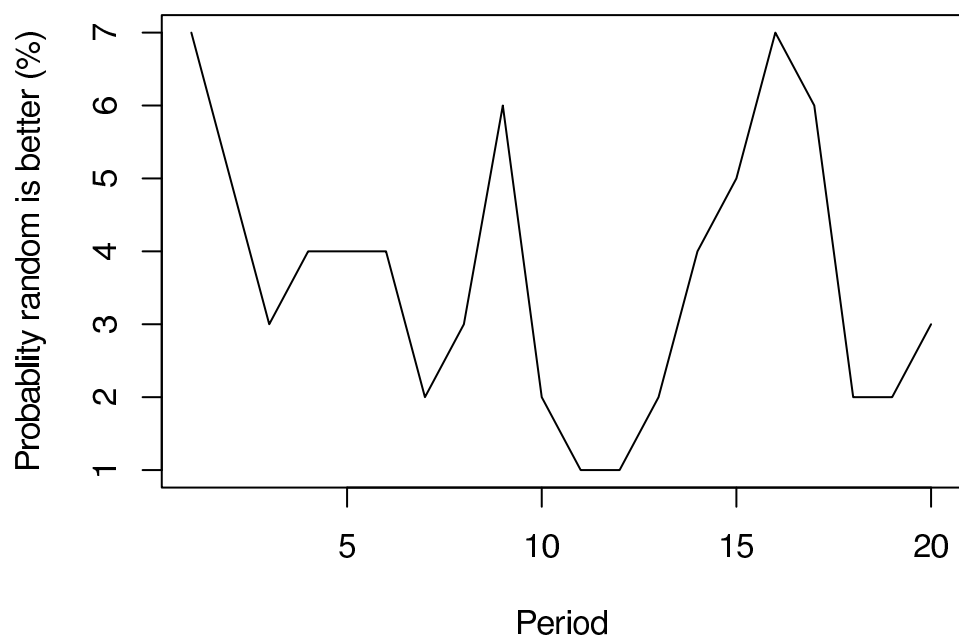
A portfolio is to be bid on, even though its composition is unknown. There will be characteristics about the portfolio which are given.

Suppose that what we know about the portfolio is that there are 45 names; the long side is \$14 million; the short side is \$9 million; the long side is 28% tech, 37% media and 35% telecom; the short side is 36% tech, 26% media and 38% telecom.

Step number one for this information is to calculate what this means for the sector constraints.

```
> long.sect <- 14 * c(.28, .37, .35)
> short.sect <- 9 * c(.36, .26, .38)
> long.sect
[1] 3.92 5.18 4.90
> short.sect
[1] 3.24 2.34 3.42
> long.sect + short.sect
[1] 7.16 7.52 8.32
> long.sect - short.sect
[1] 0.68 2.84 1.48
```

Figure 8.2: Outperformance of random period by period.



The next thing to do is to actually set up the constraints. Deciding how much leeway to allow in the constraints is undoubtedly a bit of an art. When the bounds are very tight, then the generation of the portfolios can be quite slow. Too loose of constraints can obviously give you portfolios that are far from the one of interest.

The bounds matrix in this case is set up in millions of dollars, then multiplied by a million when it is used.

```
> tmt.conb <- build.constraints(cbind(gross=sect.tmt,
+   net=sect.tmt))
> bound.tmt <- tmt.conb$bound
> bound.tmt
      lower upper
gross : media  -Inf  Inf
gross : tech   -Inf  Inf
gross : telecom -Inf  Inf
net  : media   -Inf  Inf
net  : tech    -Inf  Inf
net  : telecom -Inf  Inf
> bound.tmt[1:3,1] <- long.sect + short.sect - .1
> bound.tmt[1:3,2] <- long.sect + short.sect + .1
> bound.tmt[4:6,1] <- long.sect - short.sect - .1
> bound.tmt[4:6,2] <- long.sect - short.sect + .1
> bound.tmt
      lower upper
gross : media    7.06  7.26
gross : tech     7.42  7.62
gross : telecom  8.22  8.42
net  : media     0.58  0.78
net  : tech      2.74  2.94
net  : telecom   1.38  1.58
> rb1.port <- random.portfolio(100, prices, varian,
+ long.val=14e6, short.val=9e6, ntrade=45, port.size=45,
+ constraint.mat=tmt.conb$matrix, bounds=bound.tmt * 1e6,
+ abs.con=c(T, F))
```

caution

If you get constraints wrong (for example you forget the `abs.constraint` argument, or get the order of it wrong), then the constraints can be inconsistent and no random portfolios are found.

We can look at the summary:

```
> summary(rb1.port)
```

This shows the sizes of the portfolios (45 names); and the number of times that assets appear in the portfolios. However, appearing in a portfolio and being an important constituent of it are not the same. We can get the monetary value of assets in the portfolios, and then explore which assets are most used.

```
> rb1.val <- valuation(rb1.port, prices)
> rb1.assetval <- split(unlist(rb1.val),
+   names(unlist(rb1.val)))
```

S code note

Both `rb1.port` and `rb1.val` are lists where each component contains information on one portfolio. Object `rb1.assetval` is a list where each component contains values for a single asset.

The `unlist` function turns a list into a vector. The `split` function takes a vector and then a second vector which categorizes the values (by asset name in this case), the result is a list where each component is a vector of the values in one of the categories.

The next commands order the assets by importance:

```
> rb1.absmean <- unlist(lapply(rb1.assetval,
+   function(x) mean(abs(x))))
> rb1.avord <- rb1.assetval[rev(order(rb1.absmean))]
```

S code note

The `lapply` function loops over the components of the list. The contents of the component are fed to the given function and the corresponding component of the result is the result from the function call. So after the effect of `unlist`, `rb1.absmean` is a vector of the mean for each asset of the absolute value within the portfolios that it appears in.

The `rb1.avord` list is just a reordered version of `rb1.assetval`—the assets are in decreasing order of the mean absolute value within portfolios. The `order` function returns a vector giving the indices that would put its argument into increasing order, the `rev` function reverses its argument—hence creating a decreasing order.

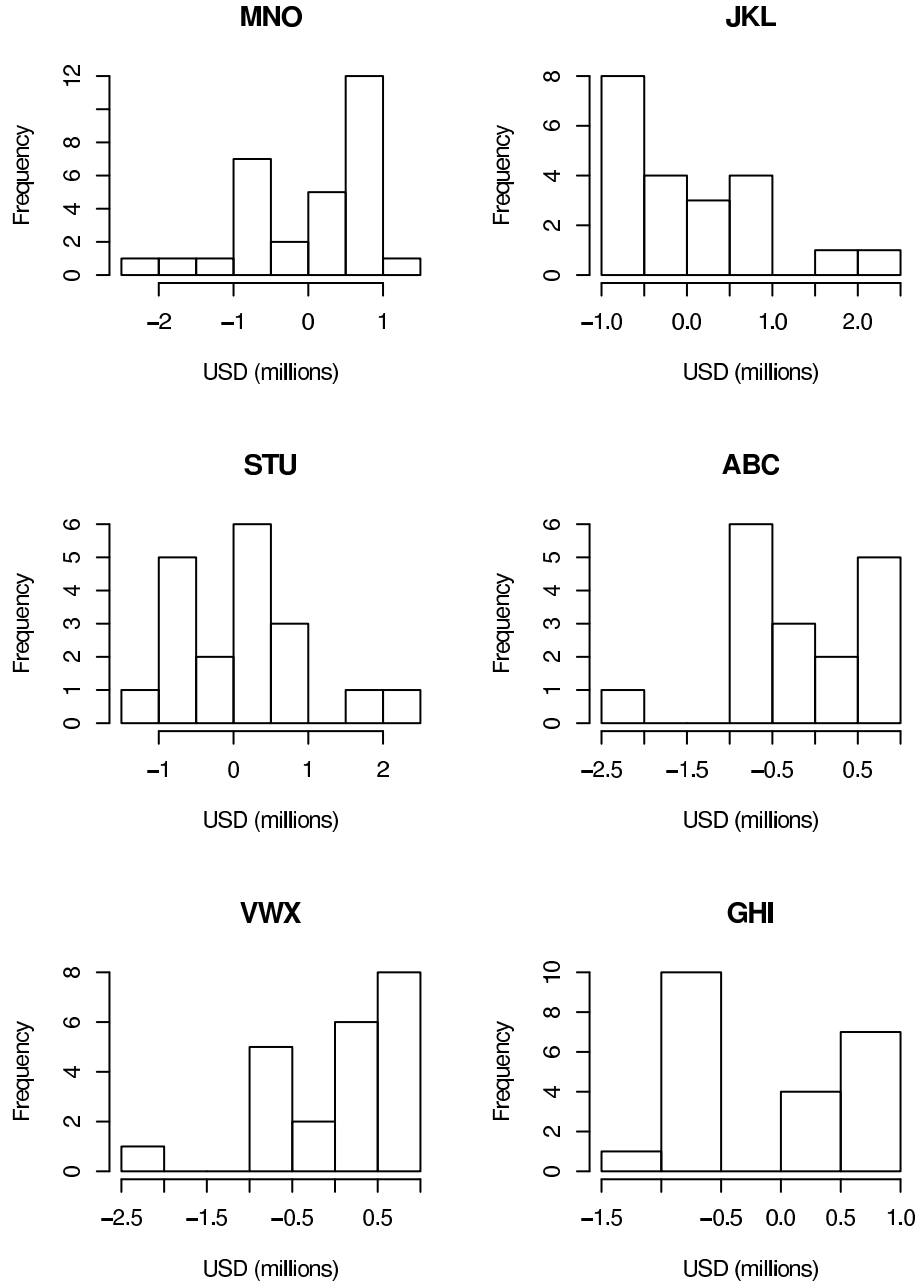
We have used mean absolute value, another likely possibility for importance would be the sum of absolute value—this latter choice would favor assets which enter more often but possibly not with as large of value.

We're now in a position to make a plot showing the distribution of monetary value for the “biggest” assets. This appears as Figure 8.3.

```
> par(mfrow=c(3,2))
> for(i in 1:6) {
+   hist(rb1.avord[[i]] / 1e6, main=names(rb1.avord)[i],
+     xlab="USD (millions)")
+ }
```

To automate the process of evaluating a portfolio, you would write one or more functions that take the information given about the portfolio and produce random portfolios, summary statistics and graphics.

Figure 8.3: Distribution of biggest assets in random portfolios.



8.5 Going Farther

Tasks that you might want to undertake:

- To review common mistakes, see Section 9.1 on page 105.
- To perform computations with multiple variances, go to Section 12.2 on page 123.

Chapter 9

Practicalities and Troubleshooting

The best question to ask after an optimization 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.

9.1 Easy Ways to Get the Optimization 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 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.

The variance matrix is singular.

While the POP 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 rather than a sample variance should be used. This problem is certainly more serious for long-short portfolios, but can also be problematic for long-only portfolios.

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.

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. The portfolio optimizer 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 to the optimizers. One check is to see if the optimizer is using the objective that you intend. You can do this by:

```
> opt.1$objective  
[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 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.

While linear constraints in asset allocation are in terms of the weights, in portfolio optimization the bounds are in monetary terms. If the bounds are not scaled, the constraints are likely to be inconsistent, in which case there will be a non-zero penalty.

If your bounds are in weights, then multiply the bounds by the desired gross value in portfolio optimization.

You do not give the `abs.constraint` or `trade.constraint` argument when needed, or reverse the meaning.

If the bounds are set for absolute constraints but `abs.constraint` is not given, then the constraints may be inconsistent. Perhaps even worse, 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 to `portfolio.optimizer` or the `buy.cost` argument to `asset allocator`. 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.

9.2 Special Optimizations

This section discusses some optimizations that are outside the usual repertoire.

Considering Higher Moments

The optimizations performed in this package depend only on the mean and the variance of the assets. The entire distribution is important though. In particular, neither skewness nor kurtosis are addressed by the optimizers. For equities and some other classes of assets this is not a problem—at least not a dramatic problem. Equities have remarkably symmetric distributions, and what skewness they have is probably close to unpredictable. Kurtosis is also unlikely to have predictability, so there is not much to do about it.

If your assets include options and other non-linear instruments or you are combining hedge funds into a fund of funds where some of the funds seem to have a skewed distribution, then an alternative optimizer is warranted. One approach is to use the Omega function—see, for example, [Cascon and Shadwick, 2004].

However, if you insist on using POP for your problem, there are some approaches you can take.

Constraining Skewness and Kurtosis

If you have predictions of skewness for each of the assets, then you can use a linear constraint with this vector to control the skewness. Note that this is not the same as constraining the skewness of the resulting portfolio. Off diagonal terms in the skewness array are assumed to be zero—rather equivalent to assuming that all correlations are zero in a variance matrix.

The first step might well be to perform your optimization without the skewness constraint and see what the skewness looks like:

```
> sk.opt1 <- portfolio.optimizer(prices, varian, alphas,
+   ...)
> violation.constraints(sk.opt1, cbind(skewness=skew.vec))
      lower upper   value nearest violation
skewness -Inf   Inf -298913.8   -Inf      NA
```

Certainly we do not want skewness to be negative, so constraining it is valuable in this case. It is easy enough to construct a sort of efficient frontier by performing a loop in which the skewness constraint is modified at each iteration.

```
> skew.con <- build.constraints(cbind(skewness=skew.vec))
> efmat <- array(NA, c(10, 4), list(NULL, c("alpha",
+   "volatility", "skewness", "penalty")))
>
> for(sc in 1:10) {
+   skew.con$bound[1,1] <- (sc-1) * 10000
+   t.opt <- portfolio.optimizer(...,
+     constrain=skew.con$matrix, bound=skew.con$bound)
+   efmat[sc, ] <- c(t.opt$alpha.val, sqrt(t.opt$var.val),
+     t.opt$violation.con[1, "value"],
+     t.opt$result["penalty"])
+ }
> efmat
      alpha volatility   skewness penalty
```

```

[1,] 7.483601  5.632706 4.383666e+00  0
[2,] 7.103521  5.680397 1.000033e+05  0
[3,] 6.651792  5.787428 2.000071e+05  0
[4,] 6.276220  6.088249 3.000091e+05  0
[5,] 6.168370  6.790503 4.000006e+05  0
[6,] 5.250250  6.746111 5.000466e+05  0
[7,] 4.358278  6.901631 6.000005e+05  0
[8,] 3.570244  7.425929 7.000040e+05  0
[9,] 3.095099  9.039130 8.000052e+05  0
[10,] 2.783357 11.929305 9.000138e+05  0

```

Recall that constraints for `portfolio.optimizer` are in terms of money, so the values for the bound are dependent not only on how much skewness you have but also on the gross value of the portfolio. In the first line of the `for` loop the lower bound for skewness is set—the upper bound is left at infinity.

The penalty is put into the output because at some point the constraint will be so strong that it can't be met. After that point the penalty will be greater than zero.

The same sort of trick can be used with kurtosis.

Lower Partial Moments

When return distributions are skewed, then variance no longer is a suitable measure of the concept of risk. One solution to this problem is to use lower partial moments—some measure of the distribution below a target value. The most common is the semi-variance, which is the second moment below the target.

Lower partial moments solve the problem, but at some cost. When you use lower partial moments, you need to decide on what to use as the target value. This is a minor annoyance in the one-period case, but becomes problematic when more than one point in time needs to be considered. Note also that if you use lower partial moments when you have a symmetric distribution, you are throwing away part of the data and hence getting more variable estimates.

It is possible to build a risk matrix using lower partial moments—see, for instance, [Scherer, 2002] and [de Athayde, 2003]. However, the “variance matrix” needs to be symmetric to work properly in POP. See page 111 for an example of symmetrizing a matrix if your matrix is not already symmetric.

Minimize Costs

If your objective is to minimize costs subject to a set of constraints, then you merely need to produce a utility that is always zero. You can do this by giving a variance matrix that contains only zeros, and not giving expected returns.

```

> zero.var <- array(0, c(length(prices), length(prices)),
+   list(names(prices), names(prices)))
>
> op.mincost <- portfolio.optimizer(prices, zero.var,
+   expected.ret=NULL, ...)

```

Non-standard Utilities

Section 8.1 points out that random portfolios can be used as an (inefficient) optimizer for any utility. See page 91 for an explanation of the process.

9.3 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 not mean-variance—it could be either maximizing the information ratio or minimizing variance. Check the `objective` component of the output.
2. You are passing in a value for the `util.table` 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 in `portfolio.optimizer` if neither the `expected.return` nor `util.table` 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 7.8.

The optimal portfolio has positions that are very small.

Use the `threshold` argument— see Section 7.8.

I'm building a portfolio and the optimizer is having problems getting the value of the portfolio right.

One possibility is that the `max.weight` 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. In a simplistic setting, a solution could be:

```
> op.ts1 <- portfolio.optimizer(prices, varian, ntrade=5,  
+   max.weight=.25)
```

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 `portfolio.optimizer`.

Use the `forced.trade` argument—see Section 7.10.

Miscellaneous Problems

I have a problem that fits into the portfolio optimization scheme except that it doesn't involve the variance.

Just give a “variance” that has the right characteristics, but put all of the values to zero. Use a command on the order of :

```
> varzero <- matrix(0, length(assetnames), length(assetnames))
> dimnames(varzero) <- list(assetnames, assetnames)
```

The optimizer is immune to strange characteristics of the variance matrix (which can be a bad thing if you feed it garbage). In particular, variances are positive (semi-)definite matrices. Most other optimizers assume or force the variance to be positive definite.

The optimizer does assume the matrix is symmetric, so trying to perform a problem with a non-symmetric matrix will not produce correct answers. In this case you should symmetrize the matrix with a command like:

```
> mat.sym <- 0.5 * (mat.asym + t(mat.asym))
```


Chapter 10

Adjusting Speed and Quality

The default control settings for the optimizers are a compromise between the time it takes to finish and the quality of the solution.

Once you are comfortable that you are doing the optimization that you want, it may be worthwhile to experiment with speed and quality adjustments. Depending on your particular optimization problem and your patience, you may want to either increase or decrease the amount of computation that is performed.

10.1 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:

- Evaluate a specific allocation relative to an efficient frontier
- See the cost of a solution
- See constraint violations of a solution
- Get expected returns, variances and so on of random portfolios
- Ensure that another program is doing the same problem
- Examine the utility of your current portfolio, or of a proposed trade

Both `portfolio.optimizer` and `asset allocator` make this easy. Give the solution 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 <- asset.allocator(aret3, avar3, risk.aver=.05,
+   start.sol=e60b40, funeval=0)
> aa.gs1$new.portfolio
      equities      bonds commodities
      0.6         0.4         0.0
```

```

> aa.gs1$utility.val
[1] 2.08
> aa.gs1$alpha.val
[1] 6.4
> aa.gs1$var.val
[1] 169.6

```

If there are constraints that the given solution does not meet, it is possible that the input starting solution might be changed. A warning is issued if the output solution is not the same as the input solution when `funeval.max` is 0 or 1. The possibility of changing the solution is a reason that it is recommended that the `ntrade` argument also be given in `portfolio.optimizer` for this type of problem:

```

> given.val <- valuation(given.trade, prices)$total['gross']
> op.gs1 <- portfolio.optimizer(prices, varian, long.only=T,
+   exist=cur.port, start.sol=given.trade, funeval=0,
+   ntrade=length(given.trade), trade.value=given.val)

```

Set the number of assets to trade to be the number that are in the given trade to help insure that it won't be changed.

caution

There is a difference in the `start.sol` argument between asset allocation and portfolio optimization.

- `start.sol` for `portfolio.optimizer` is the trade.
 - `start.sol` for `asset allocator` is the portfolio.
-

10.2 Reducing Time Use

You may want to reduce the time that an optimization takes because you are doing a large number of similar optimizations, or because you are doing very large problems that have great urgency. Strangely enough, you may also want to reduce the time used in order to degrade the quality of the optimization—an example of such a case is given in Section 6.6.

The most likely way to reduce time is to reduce `iterations.max`.

If your goal is to degrade the quality of the objective, then probably setting `funeval.max` to some smallish number is the best approach—you would need to experiment to learn what sort of range would be appropriate.

10.3 Improving Quality

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 <- portfolio.optimizer(prices, varian, ...)
> opt2 <- portfolio.optimizer(prices, varian,
+   start.sol=opt1, ...)
```

For more thorough optimization, increase `iterations.max` and `fail.iter`:

```
> opt3 <- portfolio.optimizer(prices, varian, start.sol=opt1,
+   iterations=1000, fail=10, ...)
```

Another way of doing this same thing would be:

```
> opt3 <- update(opt1, start.sol=opt1, iterations=1000,
+   fail=10)
```

S code note

If an object is a list and has a component named `call`, 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:

- `portfolio.optimizer`
 - `asset allocator`
 - `efficient.frontier`
-

The `fail.iter` argument states the largest number of consecutive iterations with no improvement that is allowed before convergence is declared. Even when `fail.iter` has been set to a large number and the algorithm has converged, there is no guarantee that the globally best solution has been found. However, in small, simple problems, the global best is often found.

When increasing `iterations.max`, note that later iterations generally take somewhat more time than earlier iterations. So multiplying the time it takes to do 200 iterations by 5 is likely to be an underestimate of the time required for 1000 iterations.

Chapter 11

Some Philosophy

This chapter highlights some theoretical topics on portfolio optimization.

11.1 The Textbook View

In the world of the textbook, returns all have a normal distribution that is constant through time. The expected returns and the variance matrix of the returns are known exactly. Given this ideal, the mean and variance are all that you need to know.

Funds are long-only so the weights are non-negative and sum to 1. Trading costs do not exist. There is no limit to the number of assets to be held in the portfolio. Hence quadratic programming gives an exact answer to the optimization problem.

There is a nice, neat package with no ambiguity— w^* is *the* answer.

What's wrong with the textbook view?

Well, pretty much everything.

Returns decidedly do not follow a normal distribution—they have long tails. There is a much higher likelihood of an extremely large return than the normal distribution predicts. A more technical way of saying this is that returns have excess kurtosis.

Since expected returns are known with hardly any precision at all, it is really unknown whether they are constant or not—but almost surely not. It is certain that the variance is not constant over time. Variances are well modeled by GARCH (see, for instance, [Alexander, 2001]).

In reality there are usually limits on the number of assets in the portfolio and the number traded.

What's been the harm?

The main harm is that fund managers have performed optimizations following the instructions in the textbook, they look at the result and say it is garbage. They then throw away optimization entirely.

What can be salvaged?

Merely adding trading costs will give useful results. This not only adds the realism that trading costs do exist, but also can help overcome the fact that expected returns and variances are not exactly known. See Section 11.5 for more on this.

While expected returns are extremely noisy estimates, many detractors of optimization probably overdo their statements on the noise in variance matrices. There certainly is noise in variance estimates, but there is enough information in them to make them quite useful.

Since returns do not follow the normal distribution, the mean and variance are no longer enough to characterize the whole distribution. However, the mean and variance still provide a useful approximation to the utility as long as the returns are symmetric, or at least not predictably skewed. This is true in many cases—equities for example. There are alternatives to mean-variance optimization when skewness does exist.

In summary, if we reduce our expectations from perfection to useful and we exercise a modicum of care, we can succeed.

11.2 Investor Optimality

The traditional use of optimization is from the fund manager's point of view. The fund manager has a mandate to have a tracking error relative to a specific benchmark that is below some value, so optimization is used to enforce that constraint while improving the expected return as much as possible.

However, it makes more sense to optimize the portfolio for the person or entity that actually owns the money. This idea is explored in [Burns, 2003c] and [Burns, 2003d].

Tracking error has probably been overemphasized. [Siegel, 2003] discusses benchmarks extensively, with a generally positive attitude towards the use of tracking error. [Burns, 2004] is more negative towards tracking error.

11.3 Bayesian Statistics

Ideas from Bayesian statistics are often useful in finance. In particular the estimation of expected returns can greatly benefit.

The core Bayesian idea is that the probability of the model parameters given the data is proportional to the probability of the data given the model parameters times the probability of the parameters.

Non-Bayesian (that is, frequentist) statistics only considers the probability of the data given the parameters. In virtually all cases a Bayesian estimate can be thought of as the frequentist estimate shrunk towards some value. How much it is shrunk and towards what value depends on the “probability of parameters” part.

For example if a frequentist estimate of the expected return for a certain asset is 10%, then a Bayesian estimate based on the same model might be 8.1%. The frequentist estimate is shrunk towards zero (either because the efficient market hypothesis says excess returns should be zero, or because the average of observed returns tends to be close to zero). If there is a lot of data and it

strongly suggests a large return, then the amount of shrinkage will be small. If there is little data or the analyst has a very strong belief that returns are close to zero, then the estimate will shrink a lot.

A more theoretical discussion of Bayes statistics is in [Scherer, 2002].

11.4 Generating Alpha

The real crux of fund management, whether optimization is used or not, is creating predictions of expected returns. When optimization is not used, there is generally no need to create numerical estimates of the expected returns—rankings are sufficient. Though strictly speaking numerical estimates are not required with optimization, it works best when estimates of expected returns are given.

[Chriss and Almgren, 2005] provide a means of getting expected returns when all you have is rankings. Their methodology is based on a solid theoretical footing. If all of the assets are in one ranking, then they provide a straightforward formula for the expected returns. In more complex situations—for example when there are rankings within sectors but no ranking between sectors—they still can come up with answers, but it is no longer a simple formula.

[Grinold and Kahn, 2000] have a discussion of generating estimates and related topics. [Scowcroft and Sefton, 2003] illustrate a Bayesian approach to getting alpha estimates.

11.5 Noisy Inputs

A well-known problem is that optimizers take their inputs as exactly known even though in reality the inputs are only estimates. In particular the expected returns are extremely variable estimates. Currently there is no universally accepted solution to the problem. The solution that has created the most heat is resampling.

Resampling Methods

[Michaud, 1998] presents the resampled efficient frontier. The idea is that the data which created the expected returns and variance are simulated numerous times and an optimization is performed with each set of simulated expected returns and variance. This is a parametric bootstrap (see, for instance, [Efron and Tibshirani, 1993] or [Chernick, 1999]). An average of the solutions is used as an improved solution. While having its good points, there are some problems as well.

The first issue is that the distribution from which to sample is not necessarily obvious. Finding a sampling distribution for the variance is generally not a problem. However, most fund managers (fortunately) do not estimate expected returns naively from observed returns, so accurately capturing the variability of the returns prediction process will be a non-trivial task.

The next issue to wonder about is whether it works or not. [Scherer, 2002] finds a few disturbing features. For example in a long-only portfolio the weight assigned to an asset can increase as its estimate of volatility increases—the opposite of what should happen.

Bootstrapping is a technique for exploring the variability of estimates rather than for improving them. As [Scherer, 2002] (page 98) states:

It is not clear, however, why averaging over resampled portfolio weights should offer a solution for dealing with estimation error in optimal portfolio construction.

Increase Risk Aversion

Unless the expected returns have been produced carefully, they will contain substantially more noise than the variance estimate. The Bayesian principle suggests that the expected returns should be shrunk (towards zero) more than any shrinking for the variance. This suggests that the risk aversion should be increased from what it otherwise would be.

Increase Trading Costs

The Bayesian point of view is that we should always shrink the naive estimate towards something when there is noise. When creating a portfolio, the resampled efficient frontier method might shrink towards something at least reasonably appropriate (though the example in [Scherer, 2002] casts some doubt on that). However, when modifying an existing portfolio, it certainly does not.

Shrinkage should be towards the existing portfolio. If we do no trading, we incur no costs, but if we do any trading, we know with certainty that we incur costs. The data needs to be convincing enough to move us to some other portfolio. Increasing the trading costs will compensate for noise in the expected returns.

Chapter 12

Advanced Features

Relatively unique features of the POP optimizers are covered in this chapter.

12.1 Dual Benchmarks

Dual benchmarks can be used 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 <- portfolio.optimizer(prices, varian,
+   long.only=T, gross.val=1e6, exist=cur.port,
+   benchmark=c("spx", "newspx"), quantile=1)
```

Typically a min-max solution is found for dual benchmarks. To get such a solution, set `quantile` to 1. One reason for min-max solutions is the unavailability of other options. There are other options in `portfolio.optimizer`, but in this case min-max is a fairly sensible objective. The min-max solution says that we want to minimize the tracking error for whichever benchmark happens to be farthest away. It is typical (though not mandatory) that the utility in the optimal solution is the same for both benchmarks:

```
> sqrt(252 * opts1$var.val)
[1] 0.009029673 0.009029442
> opts1$utility.val
[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. (To get an understanding of the mechanics, see Section 13.4 on page 135.) 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$util.tab
> utab1
      [,1] [,2]
alpha.spot      0  0
variance.spot   0  1
destination     0  1
opt.objective   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.objective 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 <- portfolio.optimizer(prices, varian,
+   long.only=T, gross.val=1e6, exist=cur.port,
+   benchmark=c("spx", "newspx"), quantile=1,
+   util.tab=utab1)
> sqrt(252 * opts2$var.val)
[1] 0.008227913 0.009735447
> opts2$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 31.

12.2 Multiple Variances and Expected Returns

Both `portfolio.optimizer` and `asset allocator` allow an arbitrary number of variances and expected returns. This section discusses a few uses of this feature. Given its novelty, there are likely to be more ideas in the future.

When there are multiple variances or expected returns, there will be multiple utilities. There is, then, the question of what characteristic of the collection of utilities the optimizer is to “optimize” on. Performing min-max optimization is the most common—that is, only the minimum utility is considered by the optimizer, and that is then maximized. The optimizers in this package allow a great amount of flexibility in what is to be optimized. In particular, any quantile of the set of utilities can be optimized. The range of quantiles that makes sense is from 1 (the min-max solution) to 0.5 (the median).

To learn how to control the use of multiple variances and expected returns, see Section 13.4 on page 135.

Rival 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 (usually) be a three-dimensional array:

```
> varmult <- array(c(vstat[vnam, vnam], vfund[vnam, vnam],
+   vmacro[vnam, vnam]), c(length(vnam), length(vnam), 3),
+   list(vnam, vnam, NULL))
```

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. In the example above, we are ensuring that the matrices we are pasting together all correspond to the same assets in the same order by subscripting with `vnam`.

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 <- portfolio.optimizer(prices, varmult,
+   long.only=T, gross.val=1.4e6, expected.ret=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 30. 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 37, then the approach is slightly different. You need to make the variance constraints argument the proper length yourself:

```
> opts.m2 <- portfolio.optimizer(prices, varian,
+   expected.ret=alphas, gross.val=1.4e6,
+   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 Section 13.4 on page 135 for explanation). 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 <- portfolio.optimizer(prices, varian,
+   expected.ret=alphas, gross.val=1.4e6,
+   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. Whether optimizing with all the variances or leaving one out for verification purposes is—at this point—probably best decided by whichever feels more comfortable to you.

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.

Preliminary research suggests that using multiple variances improves the optimization as long as the variances included are not demonstrably inferior. For example, including bootstrapped variances seems not to be advisable. There is also an indication that including separate rival expected return forecasts is less effective than averaging them into a single expected return vector. However, more research is required before definite answers can be given.

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 <- portfolio.optimizer(prices, var.market, ...)
> portfolio.optimizer(prices, var.credit, ...,
+   funeval=0, start.sol=op.cred0)$var.value
[1] 26.02844
> portfolio.optimizer(prices, var.credit,
+   expected.return=NULL, ...)$var.value
[1] 11.92467
```

Notice that the last optimization (with the credit risk) does not use any expected returns so that the objective will be to minimize the risk.

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 <- array(c(var.market[anam, anam],
+   var.credit[anam, anam]), c(length(anam),
+   length(anam), 2), list(anam, anam, NULL))
```

```
> op.cred1 <- portfolio.optimizer(prices, var.mult, ...,
+   var.constraint=c("1"=18))
```

The first command creates the three dimensional array containing the two types of risk (taking care to make sure the data line up properly). 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 `util.table` argument. We can make a simple change to the utility table from the optimization we’ve just done.

```
> op.cred1$util.table
      [,1] [,2]
alpha.spot      0  0
variance.spot   0  1
destination      0  1
opt.objective   1  1
risk.aversion   1  1
wt.in.destination 1  1
> uticred <- op.cred1$util.table
> 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 <- portfolio.optimizer(prices, var.mult, ...,
+   var.constraint=c("1"=18), util.tab=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.

Scenarios

An example of doing scenario analysis for asset allocation is in Section 5.5 starting on page 54. If scenario analysis were to be done for portfolio optimization, the process would be quite similar.

12.3 Suppressing Warning Messages

While warning messages help avoid mistakes, they can be annoying (and counter-productive) if they warn about something that you know you are doing. The `do.warn` argument can be used to suppress specific warnings. Here is an example of its use:

```
do.warn=c(cost.interc=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.

The classes of warnings that can be suppressed are slightly different between `portfolio.optimizer` and `asset allocator`.

Portfolio Optimizer Warnings

- **converged**: Convergence is declared if more than `fail.iter` consecutive iterations fail to improve the solution. If `iterations.max` iterations are performed before this condition occurs, then there is no convergence. If convergence was not achieved, you can continue the optimization with a command like:


```
> dw.opt <- portfolio.optimizer( ... )
```

Warning message:
convergence not achieved in 200 iterations (based on 3 consecutive failures to improve)

```
> dw.opt <- update(dw.opt, start.sol=dw.opt, iterations=1000)
```
- **cost.intercept.nonzero**: One or more of the cost arguments have non-zero values for the intercept.
- **value.range**: One or more of the ranges for monetary value (`gross.value` etc.) is narrow relative to the prices.
- **extraneous.assets**: One or more objects contain data on assets that are not in `prices` and hence not of use in the computation.
- **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`.
- **benchmark.long.short**: A benchmark is used in a long-short portfolio. While this can be correct, it is often not. See Section 4.9 on page 44.
- **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`.
- **tradeval.max**: The maximum `trade.value` is too large to have an effect.
- **max.weight.restrictive**: the vector of maximum weights is very restrictive. That is, the sum of the maximum weights is only slightly more than 1.

- `neg.dest.wt`: There is at least one negative value in `dest.wt`. This is almost surely wrong unless the problem is outside typical portfolio optimization.
- `penalty.size`: 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.
- `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 75.

Asset Allocator Warnings

- `converged`: Convergence is declared if more than `fail.iter` consecutive iterations fail to improve the solution. If `iterations.max` iterations are performed before this condition occurs, then there is no convergence.
- `big:problem`: If there are many assets, then `portfolio.optimizer` is likely to do better at finding the optimum.
- `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 `expected.return`.
- `extraneous.assets`: One or more objects contain data on assets that are not in `expected.return` and hence not of use in the computation.
- `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 `expected.return`.
- `neg.risk.aversion`: The risk aversion is negative.
- `no.buy.cost`: The `sell.cost` argument is given, but not `buy.cost`—meaning that there are costs for selling but no costs for buying.

12.4 Compact Variance Objects

Usually the full variance matrix is given to the optimizers. 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 uniquenesses (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 uniquenesses (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 uniquenesses.
 - *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 uniquenesses.
 - *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` 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 13

Computational Details

This chapter discusses how the computations are done in the optimizations. Though this material is not necessary for performing the optimizations, it may give you insight into any problems.

13.1 POP Constituents

In POP, optimization is split into two different functions—asset allocation and portfolio optimization. In asset allocation the answer is a set of non-negative weights that sum to 1. Portfolio optimization generally involves a selection of assets out of a larger universe, some of the positions may be short, and amounts of money and of assets are of interest as opposed to weights. This division of functionality clarifies the application to the user, and also allows the optimization routines to be specialized to the two styles of problem.

POP contains a number of S language functions. These can be divided into optimizers, variance functions, constraint functions, random generation, utilities, C programming and functions for internal use. Table 13.1 lists and describes the S functions. While this table contains about thirty functions, you are likely to only need to learn a few of them.

The object-orientation of S is one way your life gets simplified. There are, for example, four `deport` functions and four `valuation` functions in the package. All you need to know is that `deport` writes files. When you give an object to `deport`, it decides which function to actually use (or politely informs you that it is not smart enough to do what you are asking).

There are three optimization functions. The `portfolio.optimizer` function selects (a few) assets to trade. The `asset allocator` function finds an optimal allocation of all of the assets, and `efficient.frontier` uses this function to find an efficient frontier of the allocation.

Random portfolios are generated with `random.portfolio`. These are portfolios that obey constraints, but are indifferent to the utility and costs.

Variance matrices can be created with `factor.model.stat`. They can be modified with `var.add.benchmark` and `var.relative.benchmark`.

The remainder of the functions are mostly for dealing with constraints or examining the results of optimizations.

The `pop.verify` function is useful for seeing if the installation on a particular

Table 13.1: Categorization of S functions in POP.

Function	Category	Description
asset allocator	optimize	optimize asset allocation
asset allocator control	utility	control parameters for above
build constraints	constraint	create constraint objects
Cfrag.list	C code	write a fragment of C code
constraint.bnames	internal	constraint names
deport	utility	generic function
deport.efffrontBurSt	utility	export efficient frontier data
deport.portfolBurSt	utility	export optimal portfolio data
deport.randportBurSt	utility	export random portfolios
efficient.frontier	optimize	find allocation efficient frontier
factor.model.stat	variance	create statistical factor model
fitted.statfacmodBurSt	variance	variance from factor model
plot.efffrontBurSt	utility	plot efficient frontiers
pop.verify	utility	test installation
portfolio.optimizer	optimize	select optimal portfolio
portfolio.optimizer.control	utility	control parameters for above
print.portfolBurSt	utility	print method for optimal portfolio
print.randportBurSt	utility	print method for random portfolio
random.portfolio	random	generate random portfolios
random.portfolio.control	utility	control parameters for above
randport.eval	utility	evaluate random portfolios
seed.BurSt	utility	seed for random generator
summary.portfolBurSt	utility	summarize optimal portfolio
summary.randportBurSt	utility	summarize random portfolios
valuation	utility	generic function
valuation.default	utility	monetary value of a vector
valuation.portfolBurSt	utility	monetary value of a portfolio
valuation.randportBurSt	utility	random portfolio monetary value
var.add.benchmark	variance	add benchmark to variance
var.relative.benchmark	variance	make benchmark-relative variance
violation.constraints	constraint	display constraint violations

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 POP is present.

You may also use the C functions called by the optimizers directly within a C or C++ program. Before you do this, however, you should assure yourself that there is a substantial advantage to doing so. The S functions that call the C code spend a great deal of effort setting up the arguments and checking them. While your code can probably be streamlined, there will still be many lines of code to write. If this is something that you want to do, Section 13.7 on page 140 discusses some tools that are available to make this easier to do.

13.2 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 optimizers in `portfolio.optimizer` and `asset allocator` follow the convention.

The objective in these functions is the sum of three quantities: the negative of the utility, the cost, and the penalty for broken constraints. Costs are discussed in Chapter 6 while Chapter 7 covers 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 Utilities

One view is that the optimizers handle three different utilities—minimum variance, maximum information ratio, and maximum mean-variance utility. A second view is that the first two are just special cases of the last one.

In the descriptions of the utilities the weight vector w is used. In asset allocation all of the elements of w are non-negative and the sum of the elements is 1. In portfolio optimization the weights may be negative, and it is the absolute values of the weights that sum to 1—that is, the weights are the value in the portfolio of each asset divided by the gross value of the portfolio. The variance matrix is denoted V and the vector of predicted expected returns of the assets is α .

The utilities are:

- *Minimum variance*: Minimize the quantity $w^T V w$ over all vectors w that satisfy all of the constraints.
- *Maximum information ratio*: Maximize $\alpha^T w / \sqrt{w^T V w}$ over all vectors w that satisfy the constraints.
- *Maximum mean-variance utility*: Maximize $\alpha^T w - \gamma w^T V w$ 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.

Minimum variance is equivalent to a mean-variance utility with infinite risk aversion. Maximizing the information ratio is the same as maximizing the mean-variance utility for some risk aversion—the risk aversion parameter that satisfies this will be unknown for any particular problem.

The formulas above do not include trading costs. For minimum variance and maximum mean-variance the costs are clearly just added to what is there. The costs can be added as a separate term to the information ratio formula as well, but more logically costs should be a term in the numerator of the information ratio. Both versions are available, but the default is costs as a term in the numerator.

The 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, 2003c].

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.

Constraints

In addition to the utility and trading cost, the constraints are also actually part of the objective that is being minimized by the optimization algorithm. When a constraint is violated, then a penalty is added to the objective. By default the penalties are quite large so that the optimizer is quite intent on none of the constraints being broken. You can modify the penalties via the `penalty.constraint` argument.

If `opt` is the result of `portfolio.optimizer`, then `opt$result['penalty']` is equal to

```
sum(opt$penalty.constraint * opt$constraint.violation)
```

13.3 Approximate Optimality

Portfolio optimization with mean-variance utility is often regarded as a quadratic programming (QP) problem. This really is the case for simple problems like asset allocation with linear costs and only linear constraints. However, the problem jumps out of the realm of quadratic programming when common constraints for optimization are added such as the maximum number of assets to trade, round lotting, nonlinear costs, or sums of largest weights. In particular,

imposing the constraint on the maximum number of assets to trade makes the problem very difficult, and an approximation to the true optimum is all that can be guaranteed in practice.

In fact mean-variance optimization itself is a compromise for mathematical expediency. It is more natural to think of the volatility rather than the variance as the risk, but variance is the measure that allows analytic computation. In the end the two are equivalent since the same efficient frontier is generated.

The algorithms used in `portfolio.optimizer` and `asset allocator` are random algorithms (see Section 13.5 for more details). Thus each time you do the same optimization, you may get a different answer. This may be disconcerting to you relative to the certainty of a single answer from other optimizers. However, that single-answer optimizer is also giving an approximate answer (except for very simple problems)—the certainty of the answer is an illusion.

The advantage of a random algorithm is that you can choose how much computing should go into the search for the best answer. You can invest hours of computer time into a single problem if you like, or a few seconds. The discussion of trade-dependent costs on page 66 shows that there are cases where knowingly getting suboptimal random solutions is an advantage.

Since the inputs are not known exactly and prices change continuously, getting the absolutely optimal result is not necessarily a great advantage. Most problems have a large number of solutions that are close to optimal. Of course deciding what is close enough and knowing when you are there are non-trivial.

13.4 Multiple Variances and Expected Returns

The `asset allocator`, `portfolio.optimizer` and `random.portfolio` functions accept multiple variance matrices and multiple expected return vectors. The asset allocation case is simpler since it doesn't accept benchmarks (see Section 5.3 on page 51 to see how to include a benchmark), while portfolio optimization allows any number of benchmarks.

When there are multiple variances or expected returns, there is not just one objective for a given trade but a set of objectives. There needs to be a decision about how this set of objectives is turned into a single number. The largest number can be used—this yields the min-max solution (also known as Pareto optimal). Other choices could be the median of the numbers, or some other quantile.

Table 13.2 describes the pertinent arguments to `portfolio.optimizer` and `asset allocator`. The `random.portfolio` function has the same arguments as `portfolio.optimizer`. All of these arguments have defaults—you only need to give values for these if the default behavior is not what you want.

The Variance and Alpha Tables

These are not used in `asset allocator` or `efficient.frontier`.

Each of these are matrices with two rows. The first row gives the zero-based index of the variance or expected return. The second row gives the zero-based index of the corresponding benchmark. A negative number in this row means there is no benchmark.

Table 13.2: Arguments for multiple variances and expected returns.

Argument	Description
<code>util.table</code>	describes all the combinations and what to do with them
<code>var.table</code>	gives the variance-benchmark combinations (not for <code>asset.allocator</code>)
<code>alpha.table</code>	gives the expected return-benchmark combinations (not for <code>asset.allocator</code>)
<code>quantile</code>	states which quantile of the multiple objectives to use
<code>dest.wt</code>	gives weights to the multiple objectives if desired

An example of a variance table is from the dual benchmark problem on page 121.

```
> opts1$var.table
      [,1] [,2]
variance    0    0
benchmark  200  201
utility.only  1    1
```

Since there is only one variance, all elements in the first row are zero. The second row gives the zero-based indices of the two benchmarks among the assets—in this case the benchmarks are the 201st and 202nd assets. The third row is used for computational efficiency when generating random portfolios.

If there had been two variances given in this problem, then the default variance table would have been:

```
      [,1] [,2] [,3] [,4]
variance    0    1    0    1
benchmark  200  200  201  201
utility.only  1    1    1    1
```

The above matrix provides all of the combinations of variance and benchmark. If you wanted a different benchmark for each variance, you could change this to:

```
      [,1] [,2]
variance    0    1
benchmark  200  201
utility.only  1    1
```

The default behavior of the alpha table is—similar to the variance table—to give all combinations of expected return and benchmark.

The Utility Table

The utility table is a matrix that gives the combinations of variances and expected returns that are to be used and what to do with them. The dual benchmark discussion on page 121 has an example of a utility table:

```

> opts1$util.tab
      [,1] [,2]
alpha.spot      0  0
variance.spot   0  1
destination     0  1
opt.objective   0  0
risk.aversion   1  1
wt.in.destination 1  1

```

Each column corresponds to a different utility.

The first row is the expected return. If `asset allocator` is being used, then this is the zero-based index of the columns of the `expected.return` argument. In the case of `portfolio.optimizer`, this is the zero-based index of the columns of `alpha.table`.

The second row is similar to the first except that it is either the index of the variance matrix or the index of the columns of `var.table`.

The third row is the zero-based index of the destination for the utility with this combination of alpha and variance. (Here we are using “utility” to include costs and constraint penalties as well as the actual utility.) 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.

The fourth row is an indicator for the type of utility to be performed. A zero means mean-variance optimization or minimum variance, while a one means maximum information ratio.

The fifth row holds the risk aversion parameter in the case that mean-variance optimization is performed—the value is ignored if the information ratio is optimized. When the `util.table` 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 two utilities are to be computed and each put into a different destination. The first column combines the first combination of expected returns and benchmark with the first variance-benchmark combination; the mean-variance utility is computed; this utility is placed in the first destination. The second column combines the first expected returns-benchmark combination with the second variance-benchmark combination—this is placed in the second destination. Since no expected returns are given in this problem, you may think it strange that a combination of expected returns and benchmarks exists. The alpha table for this object is:

```

> opts1$alpha.table
      [,1]
alpha    -1
benchmark -1

```

These contain negative numbers, so it says that the combination to use is no expected return and no benchmark.

Controlling the Objective

In addition to entries in the utility table, the final objective is controlled by the `quantile` argument, and the `dest.wt` argument. The useful range of values for `quantile` is 0.5 (the median) to 1 (the maximum—yielding the min-max solution). 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 which is given in the utility table. 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 121.

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 54) `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.

13.5 The Genetic Algorithm

The optimizers that underlie `asset allocator` and `portfolio optimizer` are genetic algorithms. This class of algorithm allows the problem to be arbitrary. Hence the ability of these functions to have objectives, costs and constraints that are not often found in other optimizers. It also allows random portfolios to be generated with only minor modifications of the portfolio optimization code.

Many implementations of portfolio optimization using genetic algorithms—such as [Wilding, 2003]—merely employ the genetics to do better on one integer constraint (or sometimes two). These algorithms solve a series of quadratic programming problems. This process is likely to be relatively slow, and only eliminates one of the limitations of quadratic programming.

Such implementations may also be slow if they use the “standard” genetic algorithm as first proposed by [Holland, 1975]. A comparison on one simple problem of that algorithm and an algorithm somewhat similar to that used in POP is given in [Burns, 1998] on page 329. In that very simple example it took the traditional algorithm several thousand function evaluations to do as well as a few hundred function evaluations with the alternative algorithm. The difference in efficiency is likely to grow as the complexity of the problem increases.

The optimization algorithms are traditional in the sense that there are iterations, and the optimization stops when either there is a failure to improve the solution or the maximum number of iterations is performed. The difference is in what an iteration does, and the convergence criterion. In POP an iteration is a genetic population. Convergence is declared if some number of consecutive iterations fail to improve the solution.

Algorithm Components

While the POP optimizers can be described as genetic, they are not pure genetic algorithms. Here is a listing of the ingredients that go into the algorithms:

- *Totally random.* This merely creates random solutions and saves the best. The quality of the random solutions that are created is important.
- *Genetic.* This, unlike most optimization algorithms, has a population of solutions at any one time. A mating is when two of these solutions are selected and they create children that are a random blending of the features of the two parents. In the POP algorithms the best two out of the parents and the children are the solutions that survive in the population.
- *Simulated annealing.* Yet another type of random algorithm. This takes a random step from the current position. If the new solution is better, that becomes the current solution, and the random steps are centered about it. The average size of the steps decreases as the algorithm progresses—that is the “annealing” part.
- *Polishing.* This is a type of greedy algorithm. The approximate best value for each parameter is found, one parameter at a time. This is not random except for the order in which the parameters are visited.

Procedure

A population is created and optimized by combining the four components. The steps are:

- *Create the population:* This is done by filling it with random solutions, after any starting solutions are placed in the population.
- *Totally random step:* Create a series of random solutions. If a new solution is better than the worst solution in the population, then the new solution replaces the worst solution. This step eliminates the very worst solutions from the population. If too little of this is done, then later steps are less effective. Too much of this step wastes time as it is a very inefficient search.
- *Genetic step:* Perform a series of matings. If in a mating at least one child replaces a parent, then perform simulated annealing on the best child to survive from the mating.
- *Polish:* Polish the best answer in the population.

In general, several populations are optimized with each population starting with the best solution found. The number of populations used depends on `iterations.max` and `fail.iter`. The maximum number of iterations is set, but fewer than this number of populations may be performed. This happens when more than `fail.iter` consecutive populations fail to improve on the answer. In such a case the `converged` component of the result is `TRUE`.

Differences Between Asset Allocation and Portfolio Optimization

There are some differences between the algorithms for asset allocation as opposed to portfolio optimization. The portfolio optimization algorithm is more efficient for large problems. The asset allocation algorithm becomes less successful as the size of the problem grows. Problems that are larger than a couple dozen assets would probably be optimized better with `portfolio.optimizer`.

13.6 Single Trade Optimization

The genetic algorithm can not be used when there is only a single asset being traded. But this is also a very easy case, so a special algorithm is not hard to build. The algorithm evaluates the utility at `single.search` locations equally spaced over the allowable range of each tradeable asset. It then refines the size of the trade for the asset that performed best in the first stage. However, `single.search` can usually be large enough that the second stage is redundant since all possibilities have already been examined.

13.7 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 not trivial to write the program calling these functions.

Speed of execution is not a good reason to go completely to C—the execution of the S code takes a fraction of a second and so is negligible for optimizations of practical size and complexity. Perhaps the most likely reasonable reason to stay all in C is to do real-time optimization within a C or C++ environment. However, doing real-time optimization in S is quite easy as shown in Section 4.8 on page 43.

Another possible reason to use an all C implementation is if you have a gigantic universe of assets and need to conserve memory. In this case a possible means of avoiding C is to use a compact representation of the variance.

If you are determined, there is a function in the package that can help you write C code. The `Cfrag.list` function will write a file that has the C initialization of variables given in an S language list. The list that you would want to do this for is an intermediate result of an optimization. For example:

```
> op.clist <- portfolio.optimizer(prices, varian, ...,
+   intermediate="Clist")
> Cfrag.list(op.clist[-1], file="test.c")
```

The `test.c` file now holds a fragment of C code initializing most of the variables that are passed into the C function that does the optimization. This can be of use in two ways—either to initialize some variables that will always be the same (or at least mostly the same in your application), or to test your implementation as you are writing it.

13.8 Bug Reporting

Send email to patrick@burns-stat.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.
- If the problem is in `portfolio.optimizer` or `asset allocator`, then give the `version` component of an object that was created by the function. For example:

```
> op$version
           C.code           S.code
"portgen_BurSt 1.06" "portfolio.optimizer 010"
```

- 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, then also give the random seed from an object where the bug is observed:

```
> op$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

- [Alexander, 2001] Alexander, C. (2001). *Market Models: A Guide to Financial Data Analysis*. John Wiley & Sons.
- [Almgren et al., 2005] Almgren, R., Thum, C., Hauptmann, E., and Li, H. (2005). Equity market impact. *Risk*.
- [Amenc and Martellini, 2002] Amenc, N. and Martellini, L. (2002). Portfolio optimization and hedge fund style allocation decisions. *The Journal of Alternative Investments*, 5(2):7–20.
- [Bridgeland, 2001] Bridgeland, S. (2001). Process attribution—a new way to measure skill in portfolio construction. *Journal of Asset Management*, 2:247–259.
- [Burns, 1998] Burns, P. (1998). *S Poetry*. <http://www.burns-stat.com>.
- [Burns, 2003a] Burns, P. (2003a). Does my beta look big in this? Working paper, Burns Statistics, <http://www.burns-stat.com/>.
- [Burns, 2003b] Burns, P. (2003b). On using statistical factor models in optimizing long-only portfolios. Working paper, Burns Statistics, <http://www.burns-stat.com/>.
- [Burns, 2003c] Burns, P. (2003c). Portfolio sharpening. Working paper, Burns Statistics, <http://www.burns-stat.com/>.
- [Burns, 2003d] Burns, P. (2003d). Sharper fund management. Working paper, Burns Statistics, <http://www.burns-stat.com/>.
- [Burns, 2004] Burns, P. (2004). Performance measurement via random portfolios. Working paper, 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).
- [Cascon and Shadwick, 2004] Cascon, A. and Shadwick, W. F. (2004). Omega functions and Omega metrics: An introduction to Omega fund ranking. Technical report, The Finance Development Centre, <http://www.financedevelopmentcentre.com>.
- [Chernick, 1999] Chernick, M. R. (1999). *Bootstrap Methods: A Practitioner's Guide*. John Wiley.

- [Chriss and Almgren, 2005] Chriss, N. A. and Almgren, R. (2005). Portfolios from sorts. Technical report, <http://ssrn.com/abstract=720041>.
- [Dawson and Young, 2003] Dawson, R. and Young, R. (2003). Near-uniformly distributed, stochastically generated portfolios. In Satchell, S. and Scowcroft, A., editors, *Advances in Portfolio Construction and Implementation*. Butterworth–Heinemann.
- [de Athayde, 2003] de Athayde, G. M. (2003). The mean-downside risk portfolio frontier: a non-parametric approach. In Satchell, S. and Scowcroft, A., editors, *Advances in Portfolio Construction and Implementation*. Butterworth–Heinemann.
- [diBartolomeo, 2003] diBartolomeo, D. (2003). Portfolio management under taxes. In Satchell, S. and Scowcroft, A., editors, *Advances in Portfolio Construction and Implementation*. Butterworth–Heinemann.
- [Efron, 1979] Efron, B. (1979). Bootstrap methods: Another look at the jack-knife. *Annals of Statistics*, 7:1–26.
- [Efron and Tibshirani, 1993] Efron, B. and Tibshirani, R. (1993). *Introduction to the Bootstrap*. Chapman & Hall.
- [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). *CreditMetricsTM—Technical Document*. <http://www.riskmetrics.com>.
- [Holland, 1975] Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems*. University of Michigan Press.
- [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.
- [Scherer, 2002] Scherer, B. (2002). *Portfolio Construction and Risk Budgeting*. Risk Books.
- [Scowcroft and Sefton, 2003] Scowcroft, A. and Sefton, J. (2003). Enhanced indexation. In Satchell, S. and Scowcroft, A., editors, *Advances in Portfolio Construction and Implementation*. Butterworth–Heinemann.
- [Siegel, 2003] Siegel, L. B. (2003). *Benchmarks and Investment Management*. CFA Institute (for hardcopy). Full text available online at <http://www.qwafafew.org/?q=filestore/download/120>.
- [Wilding, 2003] Wilding, T. (2003). Using genetic algorithms to construct portfolios. In Satchell, S. and Scowcroft, A., editors, *Advances in Portfolio Construction and Implementation*. Butterworth–Heinemann.

Index

- abbreviation of function arguments, 92
- abs.constraint argument, 79
- allowance argument, 27, 36
- alpha generation, 119
- alpha table, 135–137
- alpha.constraint argument, 84
- alphas, *see* expected returns
- amortize cost, 65
- annualization
 - information ratio, 30
 - risk aversion, 134
- apply function, 41
- approximate optimization, 67, 134–135
- argument abbreviation, 92
- array
 - three-dimensional, 57, 123, 128
- array function, 41
- as.matrix function, 20
- asset allocation, 47–59, 131
- asset allocator function, 47–59, 123, 135, 137, 138
- assets used in optimization, 19, 27, 35
- assets, ranked, 22
- asynchronous data, 21, 106
- backtest, 22, 94
- Bayesian statistics, 118–120
- bench.constraint argument, 31, 44, 83–84, 123
- benchmark, 118
 - add to variance, 22, 51
 - alpha table, 135–136
 - argument, 44
 - constraint, 30–32, 44, 106, 124
 - dual, 31, 121–122, 136
 - long-short portfolio, 44
 - variance relative to, 51–53
 - variance table, 135–136
- beta, 90
- bidding on a portfolio, 89
- bootstrap, 119, 120
- bounds.constraint argument, 42, 77
- bug report, 141
- build.constraints function, 77
- buy-hold-sell list, 22, 32–33, 38
- c function, 31, 76, 86
- C or C++ code, 133, 140
- call component, 115
- cash, 50
- cash flow, 33–34, 39–40
- CAUTION, 21, 23, 44, 62, 63, 73–75, 100, 114, 129, 130
- cbind function, 78, 96
- Cfrag.list function, 140
- class function, 94
- close.number argument, 71
- comma-separated file, 20, 24, 93
- compact variance representation, 128–130
- compare to another program, 113
- constant distribution, 117
- constituents of package, 131–133
- constraint
 - absolute linear, 78–79
 - all, 69
 - benchmark, 30–32, 44, 106, 124
 - building linear, 77–78
 - close number, 71
 - cost, 84
 - country, 76–80, 92
 - integer, 71–72, 138
 - linear, 76–80
 - liquidity, 72–74
 - penalty for breaking, 23, 85–86, 133
 - portfolio size, 29, 37
 - portfolio value, 27–28, 35–36, 42–43

- quadratic, 86–87
- sector, 76–80, 92
- soft, 85–86
- sum of largest weights, 75–76
- tracking error, 30–32, 92, 124
- trade, 78–79
- trade number, 29, 37
- trade value, 29, 31, 37, 72
- turnover, 72
- variance, 37, 124
- violation, 79–80
- weight, 74–76
- constraint.matrix argument, 42, 77
- constraint.violation component, 134
- continuously compounded return, 21
- conventions, typography, 18
- Convergence, 127
- convergence, 115
- costs, 61–67
 - amortization, 65
 - asymmetric, 61
 - constraint, 84
 - linear, 62
 - minimize, 109
 - nonlinear, 64
 - piecewise linear, 66
 - polynomial, 63–64
 - power law, 65
 - relative to expected returns, 65–66
 - relative to noise, 118, 120
 - square root, 65
 - taxes, 63, 66
 - trade-dependent, 66–67
- country constraint, 76–80, 92
- cov.wt function, 20
- credit risk, 125–126
- csv file, 20, 24, 93
- currency, 105
- data frame, 20
- decay, time, 66
- deport function, 93
- deport functionexport function, 24
- dest.wt argument, 138
- destination, 87
 - of utility, 137
 - weight, 137, 138
- do.call function, 96
- do.warn argument, 63, 127–128
- dollar neutral, 40
- doubleconst argument, 64
- drop function, 20, 95
- dual benchmarks, 31, 121–122, 136
- efficient frontier, 47–48, 52, 55
 - resampled, 119–120
- efficient.frontier function, 47, 52, 55
- enforce.max.weight argument, 75
- error produced by S, 141
- examples, 28–34, 36–40, 48–59, 94–101
- existing argument, 107
- expected returns, 22, 30
 - estimating, 119
 - multiple, 123–126, 135–138
 - relative to costs, 65–66
 - testing, 89
- expected.return argument, 137
- exponential notation, 23
- factor model, 123
 - full, 129
 - fundamental, 123
 - macroeconomic, 123
 - simple, 129, 130
 - statistical, 20–22, 129
- factor vs factor, 21
- factor.model.stat function, 20, 130
- fail.iter argument, 115
- feasible solution, 80
- file
 - read, 19
 - write, 24–25, 93
- fitted function, 22
- force.risk.aver argument, 137
- forced trade
 - automatic, 75
- forced.trade argument, 38, 39, 84
- frequentist statistics, 118
- funeval.max argument, 93, 113, 114
- garbage, 105, 111, 117
- GARCH, 117, 125
- generating random portfolios, 89–103
- genetic algorithm, 138–140
- global data, 21, 106
- greedy algorithm, 139
- gross value
 - definition, 35

- gross.value argument, 27
- hist function, 101
- hyperpassive fund, 28
- hypothesis test, 90
- illiquid assets, 106
- import data into S, 19–20
- incomplete optimization, 67
- infinite risk aversion, 53
- information ratio
 - annualizing, 30
 - definition, 30
 - maximize, 30–32, 36, 133, 137
- inputs, noisy, 119–120
- integer constraint, 71–72, 138
- intermediate argument, 140
- intersect function, 73, 78
- inventory model, 65
- investor optimality, 118
- iterations.max argument, 115
- kurtosis, 108–109, 117
- lapply function, 95, 101
- leverage, 42
- liabilities, 44, 51
- limit.cost argument, 84
- linear constraint, 76–80
 - building, 77–78
- liquidity, 61, 72–74
- log return, 21
- long and short lists, 38
- long value
 - definition, 35
- long-only portfolio, 27–34
- long-short portfolio, 35–45, 64
- long-tailed distribution, 117
- lot
 - round, 27, 35, 72
 - size, 24
- lower partial moments, 109
- lower.trade argument, 33, 38, 72–74
- machine memory, 128
- market capitalization, 91
- market impact, 61, 66
- market neutral, 40
- market risk, 125
- matrix multiply in S, 95
- max.weight argument, 37, 75, 110
- maximize information ratio, 30–32, 36, 133, 137
- maximum weight, 75
- mean-variance
 - optimization, 32, 39, 47, 137
 - utility, 133, 137
- median optimization, 123, 135, 138
- memory of the machine, 128
- min function, 74
- min-max optimization, 57–58, 121, 123, 135, 138
- minimize costs, 109
- minimize tracking error, 28
- minimize variance, 28, 133
- missing values
 - variance estimation, 21
- monetary constraints, 27–28, 35–36, 42–43
- multiple expected returns, 123–126, 135–138
- multiple time periods, 125
- multiple variances, 123–126, 128, 135–138
- net value
 - definition, 35
- neutrality
 - dollar, 40
 - market, 40
- noisy inputs, 119–120
- non-standard utility, 91
- nonlinear costs, 64
- normal distribution, 117, 118
- not optimizing starting solution, 92–93, 113–114
- ntrade argument, 29, 33, 37, 38, 71
- numeric function, 92
- objective function, 133–134
- Omega function, 108
- operating system, 141
- optimality
 - investor, 118
- optimization
 - approximate, 67, 134–135
 - incomplete, 67
 - mean-variance, 32, 39, 47, 137
 - median, 123, 135, 138
 - min-max, 57–58, 121, 123, 135, 138

- not doing, 92–93, 113–114
- objective, 133–134
- Pareto, *see* min-max optimization
- real-time, 43–44
- single trade, 140
- wrong, 105–107
- options, 108
- order function, 101
- out.trade argument, 92
- p-value, 90
- package constituents, 131–133
- pairs trading, 40–42
- par function, 101
- Pareto optimization, *see* min-max optimization
- passive portfolio, 28–29, 121–122
- paste function, 41
- penalty for broken constraint, 23, 133
- penalty.constraint argument, 134
- plot efficient frontier, 74
- pmax function, 74
- pmin function, 74
- polishing, 139
- polynomial costs, 63–64
- pop.verify function, 131
- port.size argument, 29, 31, 37, 71
- portfolio
 - long-only, 27–34
 - long-short, 35–45, 64
 - passive, 28–29, 121–122
- portfolio.optimizer function, 27–45, 91, 94, 123, 135, 137, 138
- position
 - too small, 110
- positive semidefinite variance, 50, 111
- prices argument, 27, 35
- QP, 117, 134, 138
- quadratic constraint, 86–87
- quadratic programming, 117, 134, 138
- quantile argument, 58, 123, 138
- quotes in S, 76, 124
- R language
 - instance of S language, 14
- random algorithm, 135, 138–140
- random portfolio, 89–103, 113
- random.portfolio function, 69, 89–103, 135
- rank, variance not of full, 50, 52, 111
- ranked assets, 22
- read a file, 19
- read.table function, 20
- real-time optimization, 43–44
- rep function, 33, 38
- repeat (S construct), 44
- resampled efficient frontier, 119–120
- return
 - continuously compounded, 21
 - log, 21
 - simple, 21
- rev function, 101
- rights issue, 105
- risk
 - credit, 125–126
 - market, 125
- risk aversion
 - infinite, 53, 134
 - parameter, 120, 133, 134, 137
- risk tolerance, 134
- risk.aversion argument, 137
- rival forecasts, 123–125
- round function, 49
- round lot, 27, 35, 72
- S code note, 18, 20, 33, 38, 41, 44, 49, 55, 57, 73, 76, 78, 86, 92, 95, 96, 98, 101, 115, 123, 124
- S function
 - apply, 41
 - array, 41
 - as.matrix, 20
 - asset allocator, 47–59, 123, 135, 137, 138
 - build.constraints, 77
 - c, 31, 76, 86
 - cbind, 78, 96
 - Cfrag.list, 140
 - class, 94
 - cov.wt, 20
 - deport, 24, 93
 - do.call, 96
 - drop, 20, 95
 - efficient.frontier, 47, 52, 55
 - factor.model.stat, 20, 130

- fitted, 22
- hist, 101
- intersect, 73, 78
- lapply, 95, 101
- min, 74
- numeric, 92
- order, 101
- par, 101
- paste, 41
- pmax, 74
- pmin, 74
- pop.verify, 131
- portfolio.optimizer, 27–45, 91, 94, 123, 135, 137, 138
- random.portfolio, 69, 89–103, 135
- read.table, 20
- rep, 33, 38
- rev, 101
- round, 49
- split, 101
- summary, 23, 92
- traceback, 141
- unlist, 101
- update, 115, 127
- valuation, 24, 95
- var.add.benchmark, 22, 51
- var.relative.benchmark, 52
- violation.constraints, 79
- S language
 - includes R, 14
- scale.cost argument, 66
- scenario analysis, 54–59, 126
- sector constraint, 76–80, 92
- semi-variance, 109
- sense, making, 105
- short value
 - definition, 35
- shrinkage, *see* Bayesian
- simple return, 21
- simulated annealing, 139
- single trade optimization, 140
- single.search argument, 140
- skewness, 108–109, 118
- skill, 90
- small portfolio, 110
- soft constraint, 85–86
- split function, 101
- split, stock, 105
- stale prices, 106
- start
 - not optimizing, 92–93, 113–114
- start.sol argument, 93, 107, 113–114
- statistical factor model, 20–22, 129, 130
- statistical hypothesis test, 90
- statistics
 - Bayesian, 118–120
 - frequentist, 118
- stock split, 105
- suboptimal solution, 135
- sum of largest weights constraint, 75–76
- summary function, 23, 92
- symmetrize a matrix, 111
- tab-separated file, 20, 25
- taxes, 63, 66
- textbook view, 117–118
- three-dimensional array, 57, 123, 128
- threshold argument, 81–82
- time decay, 66
- totally random algorithm, 91, 139
- traceback function, 141
- tracking error, 118, 121
 - constraint, 30–32, 92
 - minimize, 28
- trade
 - too trivial, 110
- trade.constraint argument, 78
- trade.value argument, 27, 29, 31, 37, 39, 72
- trading costs, *see* costs
- transaction costs, *see* costs
- troubleshooting, 110–111
- turnover, *see* trade.value argument
- txt file, 20, 25
- typography conventions, 18
- uncertainty in inputs, 119–120
- unchanging utility, 110
- units of inputs, 28, 36
- universe.trade argument, 33, 38, 39, 42
- unlist function, 101
- update function, 115, 127
- upper.trade argument, 33, 38, 72–74
- util.table argument, 58
- utility
 - definitions, 133–134

- destination, 137
 - mean-variance, 47, 137
 - non-standard, 91
 - table, 57–59, 87, 136–137
 - unchanging, 110
 - zero, 109, 110
- valuation
- function, 24, 95
 - too small, 110
- value constraints, 27–28, 35–36, 42–43
- var.add.benchmark function, 22, 51
- var.constraint argument, 82–83
- var.relative.benchmark function, 52
- variance, 20–22
- adding benchmark, 22, 51
 - argument, 129–130
 - benchmark-relative, 51–53
 - compact representation, 128–130
 - constraint, 37, 124
 - factor model, *see* factor model
 - minimize, 28
 - missing value treatment, 21
 - multiple, 57, 123–126, 128, 135–138
 - non-constancy, 117
 - not positive definite, 50, 52
 - table, 135–137
 - utility minimizing, 133
 - vech representation, 129
 - zero, 50, 109, 111
- vech representation, 129
- version component, 141
- violated component, 69
- violation of constraints, 79–80
- violation.constraints function, 79
- volume, average daily, 65, 72, 73
- warnings, suppressing, 127–128
- weight
- constraint, 74–76
 - destination, 137, 138
 - sum of largest constraint, 75–76
- write a file, 24–25, 93
- wrong optimization, 105–107
- zero utility, 109, 110
- zero variance, 50, 109, 111