GREQE

GREQE–EHESS,
Centre de la Vieille Charité,
2 rue de la Charité,
13002 MARSEILLE
Tél. 91.91.92.62 et 91.90.74.40 (secrétariat)
FAX 91.90.02.27
email GREQE at FRMOP11.BITNET

Diplôme des Etudes Approfondies
en
Economie Mathématique et Econométrie

A Genetic Algorithm for Passive Management

Dirk Eddelbüttel

September 1992

Contents

List of Tables

List of Figures

Introduction

Don't expect a perfect package or miracles.
There will be enough unexpected miracles.
EBERHARD MATTES (1992)

This study links two rather recent developments from computer sciences and financial economics. During the last years, two new branches emerged from the field of artificial intelligence: neural networks (that we are not going to treat here; see, among others, Pao (1989) as an introduction) and genetic algorithms. Genetic algorithms were developed by John Holland of the University of Michigan during the last twenty years. Holland wanted to abstract the process of natural adaptation for using this concept for computer programmes. There has been a lot of interest from many different fields in genetic algorithms during the last years: engineering, optimisation, game theory, biology, image processing and many more; Goldberg (1989a) provides an overview. We will focus here on their capacity to search vast problem spaces efficiently.

Our other field of interest is passive management — an idea that emerged from the academic sphere to financial institutions practice in the early seventies and that has equally drawn more attention in the last years. Very shortly stated, the basic idea is that markets, though they may not be perfectly efficient, do not permit to find stocks with returns that are consistently higher than the market average. So one builds portfolios that shall behave like a chosen stock market index — and that guarantee hence the average market return.

The organization of the present study is as follows. We will in the first chapter gently introduce genetic algorithms on a very simple example from function optimisation. This will permit us to capture some of the basic notions and ideas behind genetic algorithms.

The second chapter then serves to discuss some of the principal results obtained in genetic algorithm analysis. We will rederive a function that describes the information processing by genetic algorithms — and thereafter discuss the question why, or why not, it might be worthwhile to operate this way. The material presented there is quite technical, so some readers might like to skip this chapter.

In the following chapter, we will terminate our presentation of genetic algorithms with some topics of genetic algorithm usage. We will see different changes that can be made to the basic genetic algorithm as presented in chapter one; all of these aim primarily at a performance enhancement for the genetic algorithm.

Thereafter, in the fourth chapter, we start considering financial economics. We will discuss topics as decision under uncertainty and its connection with portfolio analysis as well as

market models and market efficiency — these notions are useful to understand the rationale of indexation.

Indexation will be introduced in the fifth chapter. We will discuss in length the methods that are known for index fund construction.

Thereafter, in the sixth chapter, we consider the stock market index from the German Stock Exchange in Frankfurt/Main, the DAX.

Finally, we are joining all these subjects in the seventh and last chapter where we will use the genetic algorithm to find portfolios that track the DAX.

A short conclusion will then sum up our findings; an appendix that follows contains some more material on the genetic algorithm package that we have used for this application as well as the source code of all programmes that we have developed for the application.

Before we start, I would like to thank sincerely

- Prof. G. Dueck from the IBM Research Centre in Heidelberg for his help in finding this subject,
- Dr. R. Conen and A. Philippus from Deutsche Bank Research in Frankfurt/Main for some preliminary discussions and, of course, for providing the stock market data,
- Prof. K. Egle of the University of Karlsruhe for supervising this as the Diplomarbeit,
- Prof. R. Davidson of the GREQE/EHESS in Marseille for all his patience and all the discussions that helped me enormously.

All remaining errors are of course entirely mine.

Chapter 1

Introducing Genetic Algorithms

Theorists interested in optimization have been too willing to accept
the legacy of the great eighteenth and nineteenth-century mathematicians
who painted a clean world of quadratic objective functions,
ideal constraints, and ever present derivatives.
DAVID E. GOLDBERG (1989a)

1.1 Introduction

We will in a later chapter of this paper use genetic algorithms (or GAs for short) to optimise a kind of numerical function. As any optimisation method can also be seen as a search procedure over a given solution space, we will refer to genetic algorithms by saying 'optimisation method' as well as 'search' method.

Genetic algorithms are quite different from conventional optimisation methods. Taking this into account, we present them in a rather informal fashion in the next section. We introduce a 'game' similar to the one used by Goldberg (1989a) and Davidor (1991). Thereupon, we propose a solution to this game. This solution is a method derived from the essentials of genetic algorithms and we will thus become acquainted with their principal elements. The last section in this chapter summarizes the key points on genetic algorithms.

1.2 Presenting a Special Game

Suppose that you are asked to play the following game. You are opposed to a black box equipped with an input and an output (which will be described more precisely later on). The box is said to contain a function $f : R \to R$ that maps a real number d into a real number $f(d)$. You are asked to find the maximum (or the minimum if you prefer) of this unknown objective function over a specified input range $A \subset R$ where A, without loss of generality, could be set to the unit interval $[0,1]$. There is absolutely no information available on the function, i.e. whether the properties of continuity, existence of first derivatives, ... are satisfied or not. All that you have at your disposal is an output value associated to each input value that you

are trying. And to make your position even more uncomfortable: you have to 'pay' a certain amount, say, a penny, each time that you make a guess. Your objective is therefore to find with as few trials as possible a value of the objective function as good as possible.

This problem is similar to a great number of problems from the field of optimisation. It is not fully specified, it might possess multiple solutions and it might be impractical or even impossible to be evaluated systematically. Still, you might be considering one of the following strategies:

exhaustive search Try all the input values d that are possible one by one. This is possible only if the search space (the set of all possible values of d) is finite and it is worth considering only if the search space is rather small. This method will find the best value — but this simple yet inefficient method is not a very good strategy to this problem as one of the rules of the game makes you pay each time you guess.

random search Pick values of d by chance until you decide to stop. You might get away with little total expenditure, but you have neither a guarantee to find the best value nor even a guarantee to find a 'good' value (subject to a definition of 'good' and 'best' values). In the long run, this method cannot be expected to do better than exhaustive search.

heuristic search Use a 'thumb rule' to guide your search. Heuristics are mostly constructed by incorporating problem specific knowledge — which you haven't got for this problem. The objective function $f(d)$ is completely unknown so that you cannot make reference to this kind of method.

calculus based search This group falls into two sub-groups. First, indirect methods that usually set the gradient of the function to be maximized (or minimized) to zero. Second, direct methods which move along the function using local gradient information. These two methods are also excluded — no gradient information available.

So you still haven't found the nice and cheap solution method. We will therefore try to help by being more specific on the input and the output of the black box. Starting with the output, we can, without loss of generality, normalize the function to the interval [0,1]. The normalization is done in a way to ensure that 0 will be the minimum (where we do not specify whether there is a unique minimum or not) and, similarly, 1 will be the maximum value. We unveil the maximum value of 1 with the object of making the 'black box game' a little easier to play. This additional information helps to evaluate the function value returned by the black box as one can now compare this function value with the known maximum of 1. For genetic algorithm practice, it is of course not necessary to know the maximum value; the GA compares function values to the current population average and not to the (global or local) maximum. Hence, the function $f(d)$ maps from the interval [0,1] to the interval [0,1]. We suppose as well that the number of digits of the output display is finite and fix this number to, say, five digits. Concerning the input, we suppose that this value is not transmitted via a numerical keyboard but via a board of eight switches. These can be set to either 'high' or 'low' which is represented numerically by '1' and '0', respectively. The i-th switch is referred to as δ_i where δ is the vector comprising all the eight switch values. The input d is then determined by the transformation $g(\delta)$:

$$d = g(\delta) = \sum_{i=1}^{8} \delta_i 2^{-i} \ .$$

Note that this transforms the search space, i.e. the interval [0,1], that has an infinite number of points into a finite search space. Eight switch values (or 'bits') form a 'byte' with $2^8 = 256$ possible values. Taken as an example, imagine that all but the second and the fifth switches are set to zero. The string representation of this δ is `01001000`. Using the transformation $g(d)$, we get $d = 2^{-2} + 2^{-5} = 0.25 + 0.03125 = 0.28125$ as the input value for the black box.

But as any information has its price nowadays, we will have to make you get into even greater difficulties by generalizing the game. Imagine that your opponent has a 'door' to the black box. Each time that you start a new round, he might change the objective function $f(d)$ hidden in the box. So you are no longer seeking for one good result (which you could obtain by mere chance) but for a method that will lead to good results systematically. A way of doing so will be presented in the next section.

1.3 A Method to Play the Game

We will now outline a way to play the game presented in the last section. As we will see later on, this way of playing resembles the way that genetic algorithms work.

Knowing that there are 256 possible input values, we first decide to use a computer to organize our 'gambling'. The machine will help us to order the information that we possess, that is the input values that we have tried so far and the outcomes associated. Second, we decide to 'invest' in a first set of input strings that we choose by random. Hence we form a 'population' of n input strings — and as this process will be iterated, we will also use the term 'generation' to refer to a set of strings. This first generation of size, say, $n = 10$ is then evaluated one by one by the black box. In table 1.1, we show the first generation that we obtained by letting the genetic algorithm programme GENESIS (see Grefenstette, Davis and Cerys (1991) and the appendix A) choose ten strings at random for us.

Table 1.1 shows that one sample element was chosen two times. This is a result of the random sampling process; even though there is only a small chance that one of the 256 possible strings is sampled two times, this might happen. In figure 1.1, we plotted the input values of generation 1 against their output values. This shows that the sample input is biased towards the $0.3 - 0.8$ interval and that the output values yet obtained are not extraordinary, we have got only about 50% of the possible maximum value of 1.

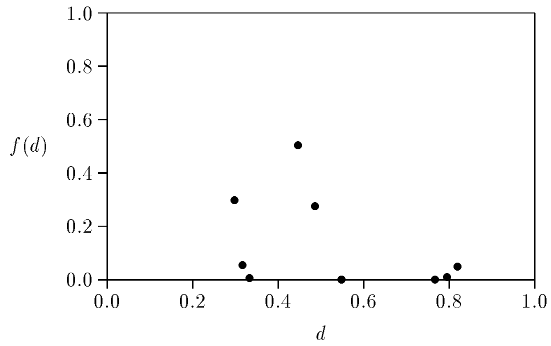
So you might have looked at table 1.1 for a moment. One compares different values for the input and the output in order to make a better guess next time. So you might have noticed that the first and the second population member have function values of roughly the same magnitude. Using common sense knowledge from mathematics, you may be thinking that there will be either a 'hill' or a 'valley' between these two values. Moreover, the best-so-far element is placed between these two values, so that there will probably a hill (which surely does not exclude the existence of an valley next to it). Concluding, you would perhaps place one of your next guesses near these three population members.

Table 1.1: The random sample at generation 1

No	Bitstring	Real Value	Output
1	01111100	0.48627	0.27341
2	01001100	0.29804	0.29544
3	01010101	0.33333	0.00464
4	11000100	0.76863	0.00003
5	10001100	0.54902	0.00000
6	10001100	0.54902	0.00000
7	01110010	0.44706	0.50209
8	01010001	0.31765	0.05315
9	11001011	0.79608	0.00783
10	11010001	0.81961	0.04649

Source: Own computations with GENESIS.
Note: We used (with the exception of the population size which was set to ten) all the default settings, i.e. Crossover Rate = 0.6, Mutation Rate = 0.001 and Random Seed = 123456789 and we adopted the elitist strategy (option 'e').

Figure 1.1: The random sample at generation 1



Source: Own computations with GENESIS.

But this kind of reasoning is too close to usual interpolation or local search methods — genetic algorithms are working in a different way. The idea thereby is to exploit similarities among the bitstrings; the genetic algorithms does not consider the 'real' value of d. Goldberg (1989a, page 13) uses the terms 'idea' and 'notion' to describe this. He calls each setting of the switches of the black box an 'idea' or prescription to solve this task of function optimising. One can then consider the substrings of a string, i.e. subgroups of switch settings which contain various 'notions' of what is important to the task. *"Viewed in this way, the population contains not just a sample of n ideas, rather, it contains a multitude of notions and ranks of those notions for task performance"*. For our example and its coding, the notions are formed by substrings and combinations of switch values. You might have noticed that the first and the second population member differ only on two of the eight positions. On the other hand, compared to the best population member, element seven, the only got three common positions. We will in the next chapter regard more closely what in genetic algorithms analysis is called a *schema*; this will be defined rigorously on page 10. Intuitively, take it for now as a similarity template for a bitstring. In a schema, we put either the actual value, i.e. a `0` or `1`, or a meta-symbol that is called *do not care symbol* which we symbolize as $*$. This means that we want to match other strings that either have the same values on the same positions that we specified by `0` or `1` or that might take any value where we have put $*$. So for the example just mentioned, we could form the schema `01*****0` that matches the three best individuals from generation 1. Genetic algorithms principally combine strings (the 'ideas') in order to promote high-performance 'notions' that build these ideas. These notions are therefore called *building blocks*.

The central idea with genetic algorithms is to exploit *all* the information, all 'notions', that we possess in the sample. We thus compute the sum of the output values and the average output value which yield 1.18308 and 0.11831, respectively. Thereafter, we calculate a *coefficient of relative fitness* for each sample element by dividing its output value by the sum of output values. As an example: this coefficient equals $0.27341/1.18308 = 0.23110$ for the first sample element.

But how does a genetic algorithm exploit this information? How does it progress from one generation to another? How does the GA find high-performance notions? The first of the *operators* that we present is *reproduction*. This is a process in which individual strings are copied according to their relative fitness. Strings with a higher value have a higher probability of contributing one or more *offspring* the next generation. This is the artificial version of natural selection. As an algorithm, it may be implemented easiest by creating a biased roulette wheel where each slot is sized in proportion to the fitness of the corresponding string. This is why this easiest form of selection is called *roulette wheel selection*. For the first bit string in generation one, this method would allocate a slice representing 23.11% of the roulette wheel size. (Recall that its coefficient of relative fitness equaled 23.11%.) Equally, the rest of the wheel will be distributed among the other population members. The computer then spins the wheel n times to create a pool of *parents* for the following generation.

These parents are then treated by the two other GA operators: *crossover* and *mutation*. Crossover, in its easiest form, the so-called *one-point crossover* , works in two steps. First, two members of the parent pool are selected randomly. Second, a position p between 1 and $l-1$ where l is the string length is selected. Two new strings of the following generation are then created by swapping all characters between position $p+1$ and l inclusively. Take as an example the strings `01001100` and `01110010` which are the elements 2 and 7 in generation one. Suppose as well that $p=4$ is selected as the crossover site. Thus the two strings `01000010` and `01111100` result and form the two offspring. This crossover operation is repeated until 'crossover rate' $\times$ 'population size' offspring are created. If the crossover rate is less then one, the new generation is completed by selecting at random the missing number of elements directly from the parent pool.

Finally, the offspring undergo mutation to form the new generation. This operator is needed because reproduction and crossover might lose some potentially useful genetic material. The mutation operator protects against such an irrecoverable loss. It takes every single bit of each string and, with a very low mutation probability, alternates its value from 0 to 1 or vice versa. Mutation is often referred to as a locally acting improvement strategy or as a hill-climber in the operations research jargon. Another possible view of mutation is to regard it as a kind of innovation generating process. The random alternation, even at low probability levels, can

introduce new 'notions' into a the current population of bitstrings. In our example, with $8 \cdot 10 = 80$ bits and a mutation probability of 0.001, we can assume that no alternation takes place.

We have thus obtained a new generation of sample elements. The idea is thus to recombine partial solutions by crossover according to their objective function values as well as an insurance against loss of information by random mutation in order to find better and better solutions to the problem. So clearly the genetic algorithm is repeating the loop of operator application until a specified termination condition is verified. For the example, we suppose of course that generation 1 does not verify this condition. So generation 1 is then evaluated by the black box and undergoes selection, crossover and mutation to form generation 2. We will accelerate a little and not print out the next generation and go, by repeating the operations described above, to generation four that we show as well in table 1.2.

Table 1.2: The random sample at generation 4

No	Bitstring	Real Value	Output
1	11001100	0.80000	0.01177
2	10010100	0.58039	0.00083
3	11100100	0.89412	0.01371
4	01111000	0.47059	0.46921
5	00010000	0.06275	0.98715
6	01010000	0.31373	0.08199
7	11001011	0.79608	0.00783
8	01110011	0.45098	0.52718
9	01110010	0.44706	0.50209
10	11001110	0.80784	0.02295

Source: Own computations with GENESIS.

So what happened to the schema `01*****0` that matched the three best individuals in generation one? It still matches three individuals in generation four, but not the most prominent ones: number four with an evaluation of 0.46921, number six with evaluation 0.08199 and element nine with evaluation 0.50209 (which survived from generation one). Obviously its average schema fitness has dropped whereas the population fitness average has more than doubled to 0.262471. So here are better schemata around that the genetic algorithm is also promoting while we were concentrating on a single. In the next chapter, we will figure out roughly how many schemata there might be processed in parallel. For now, we might try a new favourite: `0**100**` that matches again the best three.

In generation four, element five already has an evaluation of 0.98715. This compares quite favourably to the to the maximum value of 1, that we announced in order to enable this kind of comparison. This is already a rather good result for a heuristic method without problem specific knowledge. So, although we are quite content with this result, we (at least for the didactic aspect) continue for some more iterations and invest some more pence in the 'black box game'. We skip now some generations and jump directly to generation twelve that is shown in table 1.3 and figure 1.2.

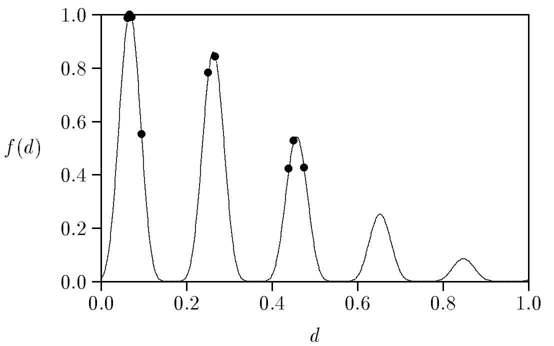
Figure 1.2 shows the twelfth generation along with the function $f(d)$ that was hidden inside the black box. We chose the function $\mathrm{F2}(x)$ from Goldberg and Richardson (1987) that we corrected in order to let the analytic form produce the function plotted by the authors. The

Table 1.3: The random sample at generation 12

No	Bitstring	Real Value	Output
1	00010001	0.06667	0.99998
2	00010010	0.07059	0.98913
3	01111001	0.47451	0.42627
4	00010010	0.07059	0.98913
5	01110000	0.43922	0.42259
6	01000100	0.26667	0.84298
7	01110011	0.45098	0.52718
8	00011000	0.09412	0.55123
9	00010000	0.06275	0.98715
10	01000000	0.25098	0.78182

Source: Own computations with GENESIS.

Figure 1.2: The random sample at generation 12 and the function $f(d)$



Source: Own computations with GENESIS. The function was proposed (in a slightly different form) by Goldberg and Richardson (1987).

corrected version then has the form

$$f(d) = \sin^6(5.1\pi d + 0.5) \cdot \exp(-4(d - 0.0667)^2) \ ,$$

a multimodal function having five peaks with decreasing magnitude.

One notes immediately that the genetic algorithm has discovered not only the biggest peak (where as well it placed forty per cent of its population, see table 1.3) but also the next two peaks. This gives a flavour of the genetic algorithms capacity to exploit the search space globally. If you consider that we now have a discrete search space (this was implied by the transformation $g(\delta)$) and if you are looking closer at the four elements put on the highest peak of $F2(d)$, you see that the GA placed one element on the best possible value, two elements on the second best and one element on the third best value!

All this happened after twelve iterations. Multiplying twelve by the population size ten yields 120 trials — which is quite a lot compared to the 256 possible discrete values. Still, this is only

a stylized and simplified example. But it seems that this is the method that you have been looking for to solve the black box game. By repeating the steps reproduction, crossover and mutation you are to locate the maxima of any function hidden in the black box — without knowing anything about these functions.

1.4 Summary

We have seen in this chapter that genetic algorithms are *"search algorithms based on the mechanics of natural selection and natural genetics"* (Goldberg (1989a), page 1). Goldberg (1989a) claims that they are *"theoretically and empirically proven to provide robust search in complex spaces"*. The central theme here is robustness as the balance between efficiency and the efficacy necessary for survival in many different environments. That is what we intended to show with the 'game' presented in the beginning of this chapter. Robustness guarantees us a relative invariance to changes in the problem specification — this is what we meant by allowing the function to change inside the black box. Complex search spaces are the ones that allow all kinds of inconvenience: discontinuity, non-existence of derivatives, multi-modality or simply a 'huge' size.

Genetic algorithms were developed to meet two goals. First, John Holland, who, along with his students at the University of Michigan, laid their foundations, wanted to abstract the adaptive processes of natural systems. Second, he wanted to build computer software that retained the important mechanisms of natural systems.

As their key elements, we note (cf. Goldberg 1989a):

- GAs work with a coding of the parameters and not with the parameters themselves,
- GAs search from populations of size n where they recombine the coded parameter information and not from single points as classical optimisation methods,
- GAs use only the objective function values and no derivatives or other auxiliary knowledge,
- GAs use probabilistic transition rules and not deterministic rules.

Chapter 2

Analysing Genetic Algorithms

The simplest operators we deal with in genetic algorithms are nonlinear;
when it comes to nonlinear systems,
many of our intuitive feelings prove to be wrong. [...]
Thus, if we are trying to develop nonlinear systems that search and learn,
we had better (at least) start off by imitating systems that work.
To do otherwise is homo sapiens chauvinism, plain and simple.
David E. Goldberg (1989d)

2.1 Introduction

In the last chapter, we have introduced genetic algorithms in an informal way. We have seen their principal ingredients and the way they work. We introduced as well the concept of schemata. In this chapter, we are going to discuss some of the results established on their performance. We will see in more detail how they work and why it might be a good idea to operate in this special way. We will therefore define more precisely the concept of schemata in the next section which will allow us to derive the so-called fundamental theorem of genetic algorithms. We then turn to statistical decision theory and will try to find out how to behave best confronted with armed bandits. Thereafter we will discuss the notion of building blocks as well as a problem especially designed to fool GAs, the GA-deceptive problem. We then try to establish an optimal population size for the genetic algorithm. A summary will conclude this chapter.

The principal reference for this chapter is Goldberg (1989a). His book is at the same time an introductory text book to the subject of genetic algorithms as well as a reference for the first results that were established.

2.2 Schemata and the Fundamental Theorem

We saw in the first chapter that genetic algorithms operate on strings that were formed over the binary alphabet. (Note that we could take another alphabet — these problems are treated

by GA theory in the field of 'coding and representation'; for numerical problems as the one that we treated in the first chapter as well as the one that we are consider later on, a binary representation is a very good choice.) Using rather simple operations as string concatenation and alternation of single positions, the GA obtained a quite good result on the function optimisation example that we considered in the last chapter. We try now to get an idea on the amount of information that is encoded in such a population of bit strings. We therefore need some notation and some definitions.

Without loss of generality, we consider strings to be constructed over the *binary alphabet* $V = \{0, 1\}$. We refer to strings by capital letters and individual characters by lowercase letter subscripted by their position. For example, $A = 01101$ may be represented symbolically as $A = a_1a_2a_3a_4a_5$. We consider a population of individual strings A_j, $j = 1, \ldots, n$ in the population $A(t)$ at time (or generation) t. We now define the meaning of a schema.

Definition 2.1 *A* schema *is a similarity template taken from the trinary alphabet* $V^+ = \{0, 1, *\}$ *describing a subset of strings with similarities at certain string positions. The '∗' is a metasymbol that is not actually processed by the GA. It matches either of the binary symbols.*

The meaning of the schema becomes clear if we think of it as a pattern matching device. A schema matches a particular string if at every location in the schema a 1 matches a 1 in the string, a 0 matches a 0, or a ∗ matches either. As an example, the string 1∗01∗ matches all the strings {10010, 11010, 10011, 11011}. On the other hand, it does not match 00010 (no match on the first position) or 11000 (no match on the fourth position). Thus, a schema gives us a powerful and compact way to talk about the similarities among finite-length strings over a finite alphabet.

We will now try to figure out how much information there might be encoded in a population and how many schemata might be formed in order to express certain notions. In the example used above where we set a string length $l = 5$, we find that there are $3^5 = 243$ different similarity templates because on each of the five positions, we may find one of the three symbols 0,1 or ∗. We can generalize this for alphabets of cardinality k: there are $(k+1)^l$ schemata that can be represented by strings of length l. So this is the total number of possible schemata for a given length and a given alphabet. Consider now a concrete string of length 5. This string can be matched by 2^5 schemata (out of the 3^5 possible schemata in the example) because each position may take on either its actual value or the do not care symbol. We say equally that this string contains these 2^5 schemata — as a schema may match a string, inversely a string represents or contains this schema. Generalizing for strings of length l: each of these strings contains 2^l different schemata. So if a string of length l contain 2^l schemata, a population of size n contains somewhere between 2^l and $n \cdot 2^l$ schemata, depending on the population diversity. So if we set $n = 10$ as in the example in the previous chapter, we see that there are at each instant t between $2^5 = 32$ and $n \cdot 2^5 = 320$ different schemata represented. (We have to correct the upper bound as we saw that there are only 243 schemata possible for this particular (k, l) tuple.) Obviously, the leverage effect becomes more impressive if we take values for real problem domains, say $n = 20$ and $l = 100$. This yields $2^l = 2^{100} = 1.27 \cdot 10^{30}$ and $n2^l = 20 \cdot 2^{100} = 2.54 \cdot 10^{31}$ schemata in a population of size $n = 50$.

Next, we define the order and the defining length of a schema so that we can calculate later on a schema equation.

Definition 2.2 *The* order *of a schema H, denoted by $o(H)$, is the number of fixed positions in the schema.*

Definition 2.3 *The* defining length *of a schema H, denoted by $\delta(H)$, is the distance between the first and last specific string position.*

Take for example $A_1 = \texttt{0*10*}$. This schema has order $o(A_1) = 3$ because there are three symbols fixed in the string: a_1, a_3 and a_4. Its defining length $\delta(A_1)$ equals 3 as well, we subtract one (first specific position) from four (last specific position). Take as another example $A_2 = \texttt{0****}$ where we obtain $o(A_2) = 1$ and $\delta(A_2) = 0$.

We are now going to consider the individual and combined effect of reproduction, crossover, and mutation on schemata contained within a population. The effect of reproduction on the expected number of schemata in the population is particularly easy to determine. Suppose at a given time step t there are m examples of a particular schema H contained within the population $A(t)$; we will write $m = m(H,t)$. As we saw in the last chapter, a string is copied according to its fitness during reproduction. In the roulette wheel selection step, a string's fitness is compared to the total population fitness; the value thus obtained determines the selection probability. More precisely: a string A_j with a fitness value f_j gets selected out of a population with total fitness value $\sum_k f_k$ with probability $p = f_j / \sum_k f_k$. So, after selecting by turning the roulette wheel n times to create a parent pool of size n from the population $A(t)$, we expect to have $m(H, t+1)$ representatives of the schema H at time $t+1$ in the population, as given by the equation

$$m(H, t+1) = m(H,t) \cdot n \cdot f(H) \cdot \frac{1}{\sum_k f_k} ,$$

where $f(H)$ is the average fitness of the strings representing schema H at time t. Using $\bar{f} = n^{-1} \cdot \sum_k f_k$ for the average of the entire population, we get

$$m(H, t+1) = m(H,t) \frac{f(H)}{\bar{f}} . \tag{2.1}$$

As an intermediate result, we state that a particular schema grows as the ratio of its average fitness to the average population fitness. This happens in parallel with every schema in a given population. All the schemata grow (or decay) according to their schema averages under the operation of reproduction alone.

It is quite interesting to alter this equation a little. Assume that a particular schema remains above average an amount $c \cdot \bar{f}$ with c as a constant, thus $f(H) = \bar{f} + c \cdot \bar{f}$. We can rewrite (2.1) as follows:

$$m(H, t+1) = m(H,t) \frac{\bar{f} + c \cdot \bar{f}}{\bar{f}} = (1+c) \cdot m(H,t) .$$

Starting at $t = 0$ and assuming a stationary value of c, we obtain

$$m(H,t) = m(H,0) \cdot (1+c)^t . \tag{2.2}$$

This is the compound-interest-equation or, generally, a geometric progression, the discrete analogue of a exponential form. The effect of reproduction is now quantitatively clear: reproduction allocates exponentially increasing numbers of trials to above average schemata (and exponentially decreasing numbers of trials to below average schemata).

This is a nice result, but reproduction alone does not do anything new to the population. Crossover is needed for a structured yet randomized information exchange. In order to quantify the effect of this operator, we have to see which kind of schemata will be destroyed by crossover. Consider the example from Goldberg (1989a, page 31). He gives the string $A = \texttt{0111000}$ along with the two representative schemata $H_1 = \texttt{*1****0}$ and $H_2 = \texttt{***10**}$. If for example position three is chosen as a crossover site, schema H_1 will be destroyed because the $a_2 = 1$ and $a_7 = 0$ will be placed in two different offspring. (The possibility that the other parent for mating has the same values on the fixed positions is *"conservatively"* (Goldberg 1989a, page 31) ignored in GA analysis.) On the other hand, H_2 is quite likely to survive as there is only one destroying crossing site. Evidently, the longer the distance between the outermost fixed positions in a schema, the more likely it is to be disrupted. This is where the defining length $\delta(A_i)$ comes in that allows to quantify the probability of schema disruption. Schema H_1 has a defining length $\delta(H_1) = 5$ and there are six possible crossover site for a string of length $l = 7$ where one of these is selected uniformly at random. So schema H_1 with $\delta(H_1) = 5$ will be destroyed with probability $p_d = \delta(H_1)/(l-1)$. More generally, we calculate a lower bound on the crossover survival probability $p_s = 1 - p_d$. Because a schema survives when the crossover site falls outside the defining length, the survival probability under simple crossover is $p_s = 1 - \delta(H)/(l-1)$, since the schema is likely to be disrupted whenever a site within the defining length is selected from the $l-1$ possible sites. If crossover is itself performed by random choice with a probability p_c at a particular mating of two parents, the survival probability may be given by the expression

$$p_s \geq 1 - p_c \frac{\delta(H)}{l-1} . \tag{2.3}$$

Assuming independence of the reproduction and crossover operations, we obtain in joining equations (2.1) and (2.3)

$$m(H, t+1) \geq m(H,t) \frac{f(H)}{\bar{f}} \left[1 - p_c \frac{\delta(H)}{l-1} \right] . \tag{2.4}$$

With both crossover and reproduction, the factor of growth (or decay) of a particular schema depends on two things: its fitness relative to the population and its defining length. Schemata with both above-average observed performance and short defining length are going to be sampled at exponentially increasing rates.

Finally, the last operator to consider is mutation, the random alternation of a single position with probability p_m. A single bit survives hence with probability $(1 - p_m)$. We defined $o(H)$, the order of schema, as the number of fixed positions in a schema H. So a particular schema survives, as the mutations are statistically independent, mutation with a probability of $(1 - p_m)^{o(H)}$, i.e. when none of its fixed positions is changed. Goldberg (1989a) (see also Bridges and Goldberg 1987) approximates this for $p_m \ll 1$ with $1 - o(H) \cdot p_m$ and concludes, while ignoring cross-product terms, for the three operators the inequality

$$m(H, t+1) \geq m(H,t) \frac{f(H)}{\bar{f}} \left[1 - p_c \frac{\delta(H)}{l-1} - o(H) \cdot p_m \right] .$$

More precisely, Liepins and Hilliard (1989) write in assuming the independence of the mutation operator:

$$m(H, t+1) \geq m(H,t) \frac{f(H)}{\bar{f}} \left[1 - p_c \frac{\delta(H)}{l-1} \right] \left[1 - p_m \right]^{o(H)} . \tag{2.5}$$

Highly fit, short defining-length schemata (called *building blocks*) are propagated generation to generation by giving exponentially increasing samples to the observed best; all this goes in parallel with no special bookkeeping or special memory other than our population of n strings. This result is so important that Holland (1975) named it the *fundamental theorem of genetic algorithms*. It is also known as *schema theorem*.

Note that schemata are called hyperplanes by some authors. To see why, consider a string of length 3 and imagine a cube in three-dimensional vector space. Each of the $2^3 = 8$ corners represents one the 2^3 strings or schemata of order 3; one could normalize the vector space by associating the string 000 with the origin. The twelve edges are schemata of order 2; for example, the edge 00∗ links the two points 000 and 001. The six planes are schemata of order 1; for example, the plane ∗∗0 covers the four points whose third co-ordinate is zero and that thus lie in the x_1x_2-plane. Finally, the schema ∗∗∗ of order 0 covers the whole cube. And counting all our components, we get $8 + 12 + 6 + 1 = 27 = (2 + 1)^3$, the number of possible schemata for strings of length three over the binary alphabet. For dimension greater than three, the notions points, lines and planes and cubes are generalized by hyperplanes. So we can think of the genetic algorithms as a mechanism that is cutting promising hyperplanes in order to find the best 'would-be-points-in-3d' in the l-dimensional space.

2.3 Considering the 2-armed and k-armed Bandit Problem

We have seen in last section that the number of promising schemata is exponentially increasing within the course of the GA. In order to see why it might be wise to allocate trials among schemata as the genetic algorithm does, we will now consider a problem from statistical decision theory, the two-armed bandit problem and its extension, the k-armed bandit problem. The answer to the question of how to gamble against these machines will shed some light on why it might be a good to allocate trials among the schemata as the GA does.

Suppose that we have a two-armed slot machine. Let us assume henceforth that we know a priori that one of the arms pays an award μ_1 with variance σ_1^2 and the other arm pays an award μ_2 with variance σ_2^2 where, without loss of generality, $\mu_1 \geq \mu_2$. We clearly want to play the arm that gives the higher expected return, but as we do not know beforehand which arm this is, we are faced with an interesting dilemma. Not only must we make a decision (or more precisely a sequence of decisions) about which arm to play, but we must at the same time collect information about which is the better arm. This trade-off between exploration for knowledge and the exploitation of that knowledge is a recurrent and fundamentally important theme in the theory of adaptive systems.

One way to approach the trade-off is to separate exploration from exploitation by first performing a single experiment and thereafter making a single irreversible decision that depends upon the outcome of the experiment. Suppose we have a total of N trials to allocate among the two arms. A strategy that might be worth trying is to allocate first an equal number of trials n trials to each of the two arms during the experimental phase (where $2n < N$). We have thus invested $2n$ trials to find the better arm and can now exploit that knowledge by allocating the remaining $N - 2n$ trials to the arms with the higher output. Supposing that the values of $N, \mu_1, \mu_2, \sigma_1, \sigma_2$ are known, De Jong (1975) calculated the expected loss

$$L(N, n) = |\mu_1 - \mu_2| \cdot [(N - n)q(n) + n(1 - q(n))]$$

where $q(n)$ is the probability that the worst arm is the observed best arm after n trials have been attempted on each machine. This probability is well approximated by

$$q(n) \simeq \frac{1}{\sqrt{2\pi}} \exp(\frac{-x^2}{2}) \quad \text{where} \quad x = \frac{\mu_1 - \mu_2}{\sqrt{\sigma_1^2 + \sigma_2^2}} \sqrt{n}\ .$$

We can see that there are two sources of loss. The first loss comes from taking the bad arm as the good one by error and attributing the remaining trials to this arm: $(N - n)q(n)$. The second arises from the exploration phase: $n(1 - q(n))$ gives the total loss from having to play the bad arm anyhow in order to gain some knowledge.

This simple strategy was easy to analyze — but there are no doubt better ways to allocate the trials to the better arm. Holland (1975) has performed calculations that show how trials should be allocated between the two arms to minimize expected loss. This results in the allocation of $n^\star$ trials to the worse arm and $N - n^\star$ to the better arm, where $n^\star$ is given by

$$n^\star \simeq b \cdot \ln\left(\frac{N^2}{8\pi b^4 \ln N^2}\right)$$

where $b = \sigma_1/(\mu_1 - \mu_2)$. Turning the equation around, we realise that the number of trials given to the observed better arm is given by the equation

$$N - n^\star \simeq N \simeq \sqrt{8\pi b^4 \ln N^2} \exp(\frac{n^\star}{2b^2})\ .$$

In other words, to allocate trials optimally, that is to minimize loss, we should give slightly more than exponentially increasing trials to the observed best arm. Unfortunately, this strategy is unrealisable, as it requires knowledge of outcomes before they occur. Nonetheless, it forms an important bound on performance that a realisable strategy should try to approach. But the reason why we presented this problem and its solution approach is to point out that the genetic algorithm that we analysed in section 2.2 comes even closer to the ideal trial allocation. The schema theorem guarantees giving at least an exponentially increasing number of trials to the observed best building blocks. And the GA has to choose between the schemata in the same way as a gambler in front of an armed bandit machine: optimise the trade-off between exploration and exploitation. In this way, the GA is a realisable yet near optimal procedure (Holland 1975) for searching among alternative solutions.

With a GA we are no longer solving a simple two-armed bandit problem; in the usual GA we consider the simultaneous solution of many multiarmed bandits. To make this more forcefully, we first consider the form of the solution to a single k-armed bandit and then demonstrate that the usual GA may be thought of as the composition of many such k-armed bandits. The form of the bounding k-armed bandit solution was first described by Holland (1973). The minimal expected loss solution to the allocation of trials to k competing arms is similar to the two-armed solution as it dictates that greater than exponentially increasing number of trials should be given to the observed best of the k arms. This result is not surprising and it connects nicely to our notions of schema processing if we consider a set of competing schemata as a particular k-armed bandit. To see this connection, we define the notion of a competing set of schemata, and then count the number and size of the k-armed bandit problems within a GA of given string length.

Definition 2.4 *Two schemata A and B of length l with individual positions a_i and b_i respectively are* competing *if at all positions $i = 1, 2, \ldots, l$ either $a_i = b_i = *$ or $a_i \neq *, b_i \neq *, a_i \neq b_i$ for at least one i value.*

As an example, $H_1 = \texttt{1**1}$ competes with $H_2 = \texttt{1**0}$ as their fourth positions have different values. But H_1 does not compete with $H_4 = \texttt{***1}$ because this last one has a do not care symbol on the first position, whereas H_1 has a fixed value on that position. For the binary alphabet V, we calculate that 2^k schemata compete over k positions that can take either 0 or 1.

We can start to see the connection to the k-armed bandit problem. Schema are competing when they have the same number k of fixed position, i.e. when they have the same order $o(H) = k$ and when they are defined over the same k positions. So the schemata are competing over the slots in the next generation as the arms of a bandit machine are competing. In order to allocate the slots in the next generation properly, we need to allocate exponentially increasing trials to the observed best schemata just as we give exponentially increasing trials to the observed best arm in the k-armed bandit problem. One of the differences between our situation in a GA and the k-armed bandit is that we have a number of problems proceeding in parallel. For example (cf. Goldberg 1989a, page 39), with three positions fixed over a string of length 7, there are $\binom{7}{3} = 35$ of the $2^3 = 8$ eight-armed bandit problems. In general, for schemata of order j over strings of length l, there are $\binom{l}{j}$ different k_j-armed bandit problems, where $k_j = 2^j$. Not all of these $\sum \binom{l}{j} = 2^l$ problems are played out equally well because crossover has a tendency to disrupt those bandits with long defining length as discussed earlier. We will now try to figure out the number of schemata that are usefully processed by the genetic algorithm.

Recall from the last chapter the counting argument that indicated that the number of processed schemata lies somewhere between 2^l and $n2^l$ in a population of size n and string length l. Not all of these schemata are processed with high probability because crossover tends to destroys those with relatively long defining length. Goldberg (1989a) states that *"the most widely quoted counting of effective schemata processing is Holland's well known, but poorly understood, $O(n^3)$ estimate"*. It means that by processing only n structures at a time, the genetic algorithm processes a number proportional to n^3 of schemata. As this is a quite impressing leverage, Holland has given it a special name, *implicit parallelism*. Still, this is only an estimate of the number of schemata processed. Goldberg (1989a), by trying to confirm this result, finally derives $n_s = 1/4(l - l_s + 1)n^3$ and concludes, since $n_s = Cn^3$, *"that the number of schemata is proportional to the cube of the population size and is thus of order $O(n^3)$"*.

So we saw by now that an exponentially increasing number is given to the promising schemata. Moreover, we discussed the k-armed bandit problem for which Holland (1973) found a solution that suggest to provide slightly more than exponentially increasing trials to the competing. So the GA does distribute the trials among the competing schemata in a rather optimal way. Finally, we considered the $O(n^3)$ estimate of the order of schema processing. This point of view is not unchallenged. We quote, as an example, from Grefenstette and Baker (1989): *"Since the counting arguments assume independence of the structures in the population, the n^3 result applies only to the first few generations of a genetic algorithm. But more important, our discussion of the k-armed bandits analogy shows that one cannot assume that the genetic*

algorithm treats every hyperplane competition as a k-armed bandit problem. Thus, more sophisticated methods are needed to quantify the parallelism of genetic algorithms."

And there are still another problems with the schema theorem. We introduced the basic genetic algorithm with one-point crossover in chapter 1. As its name indicates, its cuts schemata into two halfs. Thus, the notion of defining length is quite appropriate — as is equation (2.3) of the survival rate for a schema under one-point crossover. Unfortunately, one-point crossover is nowadays most often replaced by other crossover operators as the two-point and uniform crossover operators that we will introduce in section 3.5. And for these operators, the schema disruption analysis of equation (2.3) does not hold any longer — and the schema theorem is built upon this analysis. Similarly, roulette wheel selection is replaced by other selection mechanisms that do not automatically respect the schema theorem.

One could argue empirically that more sophisticated genetic algorithm operators are introduced only if they enhance performance — thus they must be in line with GA theory. This line of thought might intuitively be right, but it is by no means water-proof. So note that there are some theoretical results but that actual GA practice sometimes risks to be 'out-of-area', to be where there are even less theoretical results available as for the standard GA.

2.4 The Building Block Hypothesis and the Minimal Deceptive Problem

Basic schema analysis has helped to understand the behaviour of genetic algorithms. Short, low-order and highly fit schemata are sampled, recombined and resampled to form strings of potentially higher fitness. By working with these particular schemata, the complexity of the problem under consideration is reduced: instead of building high-performance strings by trying every conceivable combination, GAs construct better and better strings from the best partial solutions of past samplings. Because of the importance of highly fit schemata of low defining length and low order in the course of GAs, Goldberg (1987,1989a) has given them a special name: *building blocks*.

But Goldberg (1987) acknowledges that *"the schema theorem by itself does not guarantee convergence"* and asked if, for an arbitrary problem with an arbitrary coding, we can be sure that a simple genetic algorithm will combine short, low-order, above-average schemata to form near-optimal strings? In order to answer his question, he first refers to the *"growing body of empirical evidence"* that supports these claims for a variety of problems but states as well, acknowledging that empirical evidence is never as good as theoretic proves, but *"it does suggest that GAs are appropriate for many of the types of problems we normally encounter."* There are some more solid answers to this question. One has been provided by Bethke (1981). He has, using Walsh functions and a transformation of schemata, devised an efficient analytical method for determining schema average fitness values. This allows to identify whether, given a particular function and coding, building blocks combine to form optimal or near optimal solutions. Bethke was able to examine some sufficient conditions for GA convergence. He has shown that a problem needs misleading building blocks, that is good, short, low-order schemata must mix and form bad strings in order to beat a genetic algorithm. This class of problems that are provably misleading for the simple three-operator

GA is called *GA-deceptive*. The results suggest that functions and codings that are GA-deceptive tend to contain isolated optimal points: the best points tend to be surrounded by the worst. Holland (1987) has extended Bethke's computation to the analysis of schema averages when the population is not uniformly distributed. These transform methods are powerful mathematical tools that unfortunately have the disadvantage of being *"at least as burdensome as an exhaustive search of the underlying space"* (Goldberg 1987). So this does not provide a tool that checks whether a given problem is suitable for a GA or not.

Goldberg (1987,1989a) has tried another way to shed light on the question of what determines whether a particular problem is *GA-hard*, i.e. difficult to solve with the three-operator GA. He took up the idea of a GA-deceptive problem and tried to construct the simplest problem that should cause a GA to diverge from the global optimum. The idea is to turn the building block hypothesis upside-down: to have short low-order building blocks lead to incorrect (suboptimal) longer, higher order building blocks so that the GA will end up with a bad solution or no solution at all. The smallest possible problem, the so-called minimal deceptive problem (MDP), is a two-bit problem. In his analysis, he then discriminates on the degree of relative fitness two types of the MDP and formulates four autonomous, nonlinear, difference equations. For the type I MDP, a numerical solutions yields convergence to the global optimum as long as there is some initial proportion of the 'highest-fitness' schema in the initial population. Goldberg (1987, 1989a) claims that this can be generalized and proven, but does not give the proof. For the type II MDP, the situation is similar; but the convergence can not be assured for *any* initial population. However, Goldberg is able to give a sufficient condition for convergence. Goldberg concludes that even GA-deceptive problems are not usually GA-hard (i.e. does not usually diverge from the global optimum). We cite from Goldberg (1987): *"it has been shown that under a wide range of starting conditions, the genetic algorithm converges (in expected proportion) to the global optimum regardless of crossover rate, and regardless of schema defining length."*

2.5 Summary

In this chapter, we have seen the cornerstones of genetic algorithm analysis:

- the schema theorem showed that high-performance, short defining length, low order schemata receive at least exponentially increasing numbers of trials in successive generations by using the basic genetic algorithm,
- the analysis of the two-armed and $k-$armed bandit problems illustrated that giving slightly more than exponentially increasing trials is an optimal strategy for the $k-$armed bandit problem — and that the GA can be viewed as such a problem, which makes that his operations seem to be quite rational,
- the implicit parallelism was discussed: although the GA only processes a population of size n, he does take into account a far greater number of schemata at any moment; the estimate of $O(n^3)$ schemata is often given but is not unchallenged,
- thus the building block hypothesis is formulated: a GA reduces the complexity of arbitrary problems by speculating on many combinations of the best partial solutions contained in the current population and

- the analysis of the Minimal Deceptive Problem showed that even problems designed to fool the GA (GA-deceptive) are not GA-hard, i.e. they convergence to the global optimum for most initial conditions.

This chapter might not leave the reader with the feeling that this is a field of research where the answers are clear-cut. And that might indeed be the case — it is a discipline that is relatively young and prospered in the last ten years. The work on GAs is still continuing and the number of papers is steadily growing. Recently, Eiben, Aarts and Van Hee (1990) have published a book in which they generalized the genetic algorithm to an abstract genetic algorithm. Using Markov chain analysis, they were able to prove theorems that give conditions implying that an optimum is reached with probability one. This work is still recent, but maybe the next introduction to genetic algorithms that the reader touches can provide more solid answers to the question of genetic algorithm convergence.

Chapter 3

Using Genetic Algorithms

The proof is in the empirical pudding.
Gilbert Syswerda (1989)

The proof of the pudding is in the eating.
British proverb

3.1 Introduction

In the last two chapters, we have seen what genetic algorithms are about and what arguments have been presented as an attempt to confirm analytically their empirical success. We will now turn towards different operators, algorithms for GA parts and parameter settings that can be used to ameliorate the results achieved by GAs. We will use the classification of the principal ingredients of genetic algorithms by Davis and Steenstrup (1987) and devote a section to each of the five components representation, initialization, evaluation function, genetic operators and, last, parameter values.

3.2 On Representation

Genetic algorithm theory and practice started with John Holland. He used bit strings — lists filled exclusively with zeros and ones. As we saw in chapter 1, this allows a straightforward representation for a function optimisation problem with one (numerical) argument. And for more than one argument, we can link the representations of each of these arguments. So if we want, say, to maximize a function as $f(x, y, z)$ where each variable represents a real number that is encoded in ten bits, we simply get strings of length thirty by concatenation of the individual strings. We obtain hence an evaluation function that still has one single argument — that is a vector compromising in this case three elements.

But why are we using the binary coding in the first place? Goldberg (1989a) compares the coding of the integer set $[0, 31]$ by a 32-letter alphabet (i.e. the 26 letters $\{A, \ldots, Z\}$ plus the 6 digits $\{1, 2, \ldots, 6\}$) and alternatively by binary string of length 5 (recall that $2^5 = 32$). He presents the two in a table and it is evident that the binary coding that takes up five "columns"

gives more chances to figure out similarities — the schemata. The nonbinary coding with cardinality 32 that uses only one column does not allow this. In (1989d), Goldberg stated that *"In general, the number of schemata S per bit of information in a string coded over an alphabet of cardinality k may be calculated as $S = {}^{\log_2}\sqrt[k]{k+1}$. Maximizing this expression with respect to k and holding the constraint that we must have alphabets of cardinality $k \geq 2$, we obtain an optimal cardinality $k^\star = 2$."* As the binary alphabet offers the maximum number of schemata per bit of information, Goldberg (1989a) postulated the *principle of minimal alphabets*: the user should select the smallest alphabet that allows a natural expression of the problem. So for any problem that can be expressed numerically, one is on the safe side by taking the binary alphabet.

Goldberg (1989a) also postulated the *principle of meaningful building blocks*: the user should select a coding so that short, low-order schemata are relevant to the underlying problem and relatively unrelated to schemata over other fixed positions. This means that the distance between related bit positions should be as small as possible. Consider table 3.1.

Table 3.1: Comparison of binary-coded and Gray-coded integers

Integer	Binary	Gray	Integer	Binary	Gray
0	0000	0000	8	1000	1100
1	0001	0001	9	1001	1101
2	0010	0011	10	1010	1111
3	0011	0010	11	1011	1110
4	0100	0110	12	1100	1010
5	0101	0111	13	1101	1011
6	0110	0101	14	1110	1001
7	0111	0100	15	1111	1000

Source: Goldberg (1989a), page 100.

As the table indicates, adjacent integers differ only by a single bit under Gray-coding, i.e. their Hamming-distance is one whereas the Hamming-distance under binary coding can be as big as four (e.g. between the integers 7 and 8).[1] So a slight change of the parameter that might be "desired" can translate into a complete change in the string. This problem of non-adjacent coding for adjacent integer values is called Hamming-cliffs, signalling that local search routines might have difficulties in passing these cliffs. Caruana and Schaffer (1988) investigated in the problem of representation and found that Gray-coding often leads to better results than binary-coding for genetic algorithms involving numerical parameters.

Note that other problem domains lead to other coding problems. Consider, as an example, the problem of the travelling salesman where a route of minimal length covering all cities (from a given set of cities) at least one time has to be found. One could simply associate a number to each city and take the route length as the objective to be minimized. But simple crossover and mutation will then certainly produce degenerated tours — where say, the first and the third city figure several times and the second city does not figure at all. One does need a crossover operator that not only cuts and thereby eventually creates such a new individual

[1] The Hamming-distance for two different words of the same length and under the same code was defined by R.W. Hamming as the number of different positions between the two words (Engesser 1989). For the Gray coding, we have found neither a proper definition nor an algorithm for this coding in any of the papers that refer to it.

that does no longer represent a valid tour, but a more sophisticated crossover operator that combines valid sub-tours to new tours.

Generally, other representations might necessitate other crossover operators. Scheduling, the assignment of a production plan for a factory plant or of teachers to classes, is another example that requires an ordered representation. See Goldberg (1989a) among others.

3.3 On Initialization

Initialization of the population is carried out by a procedure that (in the case of a binary representation that we assume henceforth) "throws un unbiased coin" and thus produces random strings of the desired length. This is the standard and it does pretty well on most problems — provided that the algorithm for generating pseudo-random figures is correct. As we have seen so far, the stochastic transition rules of GAs require a generator for uniformly distributed numbers. Examples are the probabilities of executing the operators crossover and mutation, selecting a crossover site or selecting parents for mating. In order to ensure that these uniformly distributed figures are correctly computed, the GENESIS implementation package (see the appendix A) comes with its own generator.

But one can as well introduce some "guesses" or known solutions as an initialization for the first generation. Davis (1991) advocates the use of hybrid systems, that is the fusion of a method adapted to the problem domain and a genetic algorithm. Davis (1989) claimed that *"it is a truism in the expert system field that domain knowledge leads to increased performance in optimisation* [...] *Hence, if we resist adding knowledge to our genetic algorithms, they are likely to underperform nearly any reasonable optimisation algorithm that does take account of such domain knowledge"*. To put this into perspective, note that Davis worked mainly on industrial applications that are quite different from the simple function optimisation examples that we considered that far. Davis (1989) gives some new GA operators as well as adapting probabilities for their use.

One possible way of creating hybrid systems is by optimising via a two-stage-process. During the first stage, one tries to find some reasonable values that one tries to improve during the second stage. Either of these two steps can be executed by a heuristic method, i.e. a method that has a high probability but no guarantee to find a solution to a given problem (Engesser 1989). Suppose that each call of the heuristic procedure used to find a first population of size n gives one approximation of the solution — so that n calls will give the desired initial population for the genetic algorithm that tries to improve in the second stage.

Still, hybrid genetic systems might be used more fruitfully by turning the two stages around. Grefenstette (1987b) concludes, after having studied the travelling salesman problem (TSP), that *"genetic algorithms are not well suited for fine-tuning structures which are very close to optimal solutions. Instead the strength of GAs is in quickly locating the high performance regions of vast and complex search spaces."* So one would use the GA to generate a pool of good start solutions and then use a problem specific heuristic to reach the ultimate points. This is in fact an approach that is used in the Operations Research literature for the TSP; a common solution is to use an "opening heuristic" to generate start values followed by an "ameliorating heuristic" to improve performance (Kramer and Morlock 1988). Among the

GA papers on heuristic use, see Liepins, Hilliard, Palmer and Morrow (1987), Suh and Van Gucht (1987), Jog, Suh and Van Gucht (1989) for a further discussion.

3.4 On Evaluation Functions

The evaluation function is the interface to the real-world problem. The genetic algorithm presents a bit string — and the evaluation function returns a value for the objective function. It is one of most attractive points on genetic algorithms that absolutely nothing about the problem domain is taken into account by the problem solving procedure, the GA. Grefenstette (1987b) calls the GA a *"powerful weak method"*. He thereby refers to weak methods that, with making only few assumptions on the problem domain, are applicable to a broad range of problems. Yet, they are often not very powerful whereas GAs efficiently exploit the search space. One can take the same genetic algorithm and optimise two absolutely distinct problems whereas performing heuristics normally require that an expert finely tunes and adopts the heuristic to the problem domain. Genetic algorithms, on the other hand, are *generic*; ones you have a nicely working GA, you attach it without further ado to another problem by simply switching to another evaluation function. Still, for "high performance" solutions, problem-specific knowledge or fine-tuning might be necessary.

We have not yet addressed the possibility of negative fitness function values. This can be dealt with by providing a transformation of the evaluation function $f(d)$. Instead of passing the value $f(d)$ to the GA, we transform with a suitable function $h(.)$ and pass $h(f(d))$. For negative values of $f(d)$, we set $h(f(d)) = \alpha f(d) + \beta$ and initialize β with the smallest possible value for $f(d)$, multiplied by -1, and $h(f(d))$ will always map onto non-negative values. The value of α will either be 1 or -1 as to allow to switch between minimization and maximization problems.

One of the problems in genetic algorithm practice is *premature convergence*, i.e. the early convergence towards a local optimum. At the start of a GA run it is common to have a few extraordinary sample elements among mediocre colleagues. So if the roulette wheel is sized according to the simple $f/\sum f$ rule, these outraging sample elements rapidly take over a great part. There is a trade-off between convergence speed and the ability to locate the global optimum in the problem domain; high speed might entail getting stuck around local optimal points. One of the answers proposed to overcome this difficulty is *evaluation function normalization*, that is by choosing appropriate values for α in the formula given above. We will discuss this further in section 3.5.

Genetic algorithms can handle constraints as well. Equality constraints can be expressed directly in the underlying model. For inequality constraints, the most common way is to express them as penalties as to get some information out of the degree of violation. Goldberg (1987a) thus writes the problem

$$\text{minimize } f(x) \quad \text{subject to } g_i(x) \geq 0 \quad i = 1, 2, \ldots, n \,,$$

where x can be either a scalar or a vector by introducing the penalty function Φ and the penalty coefficient r as

$$\text{minimize } f(x) + r \cdot \sum_{i=1}^{n} \Phi\left[g_i(x)\right] \,.$$

He states that one usually takes the square of the violation for Φ and that one chooses r separately for each constraint. Constraints can also be implemented by creating decoders of the representation that avoid sample elements that violate the constraints. Each of these methods has its advantages and its disadvantages — and most often are they problem specific so that fine-tuning is necessary. See for example Richardson, Palmer, Liepins and Hilliard (1989) and Michalewicz and Janikow (1991) for a more detailed discussion of constrained optimisation with GAs.

3.5 On Genetic Operators

The first of the GA operators is reproduction. In the reproduction phase, the individual number of offspring for each sample element is determined according to its performance, i.e. its fitness function value. The operation actually consists of two parts, the *selection*, where the expected number of offspring is calculated, and the *sampling*, the conversion of the expected values to discrete numbers of offspring.

Selection is closely related to problem of premature convergence mentioned in section 3.4. Whitley (1989) advocated for rank-based selection in order to control convergence speed more appropriately. He argued that the primary factors in genetic search are *population diversity* and *selective pressure* which are inversely related. He then noted that the fitness proportional reproduction (i.e. sizing roulette wheel slots on raw fitness function values) can lead to problems where search stagnates because it is lacking selection pressure or that search converges prematurely because it is narrowed too quickly. These problems are both related to scaling. Suppose that a GA is maximizing a function where values range between 1100 and 100. The initial pressure on the top-ranked individual is, with a sample average of 550, $1100/550 = 2$. Later in the search, the range may be between 1000 and 1100 with average 1050. The pressure on the best individual is thus fallen to $1100/1050 = 1.05$. Search is thus likely to stagnate because the performance differences between sample elements are relatively small and the more promising ones do not obtain enough attention. Whitley (1989) presented the idea of ranking the individuals according to fitness, which other authors had used beforehand. He gave an adapted version of the schema theorem in order to let ranking harmonize with GA theory. Whitley then compared a GA with his ranking algorithm to the standard GA on the De Jong (1975) test suite as well as on neural network connection weight optimisation.

Whitley (1989) concluded: *"For bias up to and including 2.0, a linear function is used to allocate reproductive trials. In the neural net experiments reported here a selective bias of 1.5 was found to give the best overall results. [...] A bias of 1.5 implies that the top ranked individual in the population is 1.5 times more likely to reproduce (on one reproductive cycle) than the median individual in the population."*

We now consider the sampling side of reproduction where the problem consists of according integer numbers of offspring 'best' according to their expected values obtained in the selection stage. In an influential paper, Baker (1987) introduced an optimal sampling algorithms. Baker stated that none of the known sampling algorithms provided both zero bias (between sampling probability and expected value) and minimum spread (the range of possible values for the actual number of offspring). He discussed the algorithm efficiency as well and stated that for

example the roulette wheel selection algorithm has a complexity of $O(n^2)$ whereas the other parts of the GA are within $O(l \cdot n)$ or better. Baker (1987) then presented two new sampling algorithms. The first, called "remainder stochastic independent sampling (RSIS)", can be executed partially in parallel while having low bias, minimal spread and complexity $O(n)$. The second, called "stochastic universal sampling (SUS)", has zero bias, minimum spread, complexity $O(n)$ and only four lines of code. But as the code is rather complex, we only give Baker's description: *"On a standard spinning wheel, there is a single pointer which indicates the "winner". SUS is analogous to a spinning wheel with n equally spaced pointers. Hence, a single spin results in n "winners". Since the sum of the population's expected values = n, the pointers are exactly 1.0 apart. Thus an individual is guaranteed $\lfloor$expected value$\rfloor$ in samples and no more than $\lceil$expected value$\rceil$. Hence, SUS has minimum spread. Furthermore, in a randomly ordered population, an individual's selection probability is based solely on the initial spin and the magnitude of his expected value. Hence, SUS has zero bias."* Baker's optimal sampling algorithm SUS is now the uncontested standard in sequential GA applications.

The problem of premature convergence was further studied by Eshelman and Schaffer (1991). They argued that population diversity must be maintained at three different stages. The first stage is the selection of the mates. Eshelman and Schaffer (1991) proposed a new mechanism named *incest prevention*. After the random selection of mates, the mating takes place only if the Hamming-distance of the mates exceeds a certain threshold. The latter is lowered towards the end of a run when the population converges. The second stage is the actual creation of children. Eshelman and Schaffer presented a new crossover operator that is in a way a synthesis of the one-point, two-point and uniform crossover operators that we are going to present shortly. The third stage is the replacement of parents by children for the next generation. Eshelman and Schaffer adopted a variant of the elitist schema that we will introduce in section 3.6. The empirical result of Eshelman and Schaffer were encouraging — their GA outperformed the standard GA. Incest prevention had a statistically significant effect on twelve of the thirteen test functions (that were basically the ones that we will discuss in 3.6); their new crossover operator improved performance significantly as well.

The second genetic algorithm operator is crossover. Historically, all started with the *one-point crossover* that we presented in chapter 1. De Jong (1975), in a study that we refer to in more detail in section 3.6, concluded that *two-point crossover* is superior on most problems. As its name indicates, two-point crossover is a rather straightforward variation. Instead of choosing one crossover site and exchanging all the information to its side, one chooses two crossover sites and exchanges all the information between the two sites. Two-point crossover is now used by the vast majority of GA applications.

But as there are still schemata that can not be translated by two-point crossover, Syswerda (1989) introduced *uniform crossover* and compared it to one-point and two-point crossover for function optimisation problems. The uniform crossover operates by using a 'mask', that is a string of the same length as the parents (and thus the offspring). The crossover mask is created by throwing an unbiased coin l times. This yields a string of length l over the binary alphabet. Uniform crossover then constructs the offspring in a loop. The algorithms regards at each position p with $p = 1, 2, \ldots, l$ the value of the crossover mask. If the value is a zero, the algorithm places the actual bit p of parent i at the same position into the corresponding child, i.e. from parent i to child i with $i = 1, 2$. But if the mask value is one, the actual bit will put into the other offspring: child $1 - i$ receives the information of parent i.[2] Using schema analysis and empirical tests on the De Jong test suite (see section 3.6), Syswerda concluded that two-point crossover performs consistently better than one-point crossover and that uniform crossover outperforms both, with one exception. Still, it might not be appropriate to generalize Syswerda's results as he obtained them on a variant of the GA — the *steady state genetic algorithm* (SSGA). In the SSGA, after initialization, a loop is executed until the termination condition is reached. In the loop, two parents are selected and crossover and mutation are executed to create two offspring. These two offspring are inserted into the generation while two others are replaced. These latter are selected by *inverse ranking*, that is, by starting with the worst members. Syswerda set the selection probability to 10 divided by the population size. The the loop starts again. Syswerda claims that the SSGA allows for more aggressive learning rate (i.e. faster convergence) while schema fitness versus percentage in the population works out as the fixed point of the system.

As we have mentioned above, Eshelman and Schaffer (1991) successfully presented another crossover operator that they called *HUX crossover* which stands presumably for half-uniform-crossover. HUX crosses half of the bits that differ between the two parents at random. This crossover operator is thus a compromise between uniform and two-point crossover.

Finally, mutation forms the last genetic algorithm operator. There is not a lot that one can change with mutation — apart from finding the appropriated mutation rate. We will refer to this value along with the other parameter settings in section 3.6. For the binary representation, mutation switches to the other symbol whereas more complicated representations require more complicated mutation mechanisms. But as we are not investigating in other kinds of representation, we will leave this question apart.

3.6 On Parameter Values

We already mentioned the problem of premature convergence in section 3.4. The setting of the genetic algorithm parameters, i.e. population size n, crossover rate p_c and mutation rate p_m, is often considered as primordial in order to avoid getting lost on points that are locally optimal. As an example, Goldberg (1989c) wrote: *"Choosing the population size for a genetic algorithm is a fundamental decision faced by all GA users. One the one hand, if too small a population size is selected, the genetic algorithm will converge too quickly, with insufficient processing of too few schemata. On the other hand, a population with too many members results in long waiting times for significant improvement, especially when evaluating of individuals within a population must be performed wholly or partially in serial: the populating is too large to get enough mixing of building blocks per unit of computation time."*

The first milestone on parametrization was the dissertation of De Jong (1975) where he compared genetic algorithms with the best gradient techniques. He used a test suite of five functions that was designed to take into account the characteristics continuous ↔ discontinuous, convex ↔ nonconvex, unimodal ↔ multimodal, quadratic ↔ nonquadratic, low-dimension ↔ high-dimension and deterministic ↔ stochastic which prove more or less difficulty for gradient techniques. De Jong (1975) defined the following performance measures.

[2] For the binary values 0 and 1, the expression $1-i$ is equivalent to not i, the boolean negation of the value of i.

Definition 3.1 *The* on-line performance *is the average of all evaluations of the function $f(d_{t,i})$ where t is the generation index and i is the sample element index:*

$$\textit{on-line}(T) = \frac{1}{Tn}\sum_{t=1}^{T}\sum_{i=1}^{n} f(d_{t,i}) \; .$$

Definition 3.2 *The* off-line performance *is the average of the evaluations of the generation-best d_t^* from each generation t:*

$$\textit{off-line}(T) = \frac{1}{T}\sum_{t=1}^{T} f(d_t^*) \; .$$

In words, on-line performance is the running average of *all* evaluations so far whereas off-line is the running average of *all generation best* evaluations so far. To do well on on-line performance, a search algorithm must quickly decide where the best values lie and concentrate its search there. De Jong (1975) finally provided the set of parameters

populations size n size between $n = 50$ and $n = 100$,
crossover rate $p_c = 0.6$ and
mutation rate $p_m = 0.001$,

that produced good results on the test suite.

De Jong (1975) studied as well the impact of the *elitist strategy*, the *generation gap g* and the influence of several crossover points. An elitist strategy takes care that by progressing from generation t to $t+1$, the best element d_t^* is included in the generation $t+1$. If the usual operators did not create a sample element identical to d_t^*, it is included by the elitist strategy in generation $t+1$. De Jong found that the elitist strategy improved, for the unimodal function, both on-line and off-line performance. He suggested that this strategy improves local search at the expense of global perspective. A generation gap g, as opposed to the non-overlapping generations reproduction from chapter 1, ensures that $n \cdot g$ elements are selected each generation for the usual reproduction process whereas $n(1-g)$ elements are picked at random to take part in the following generation. The analysis of multiple crossover points let De Jong recommend the two-point crossover that we have presented in section 3.5.

Eleven years later, Grefenstette (1986) used a GA to solve the control parameter problem, that is he built a meta-GA that was searching the space of control parameters for a second GA that was executing the functions from the De Jong (1975) suite. This apparently bright idea brings the advantage of efficiently exploring the vast meta-problem space of control parameters at the cost of not providing insight into the sensitivity of the results to variations of the parameter values. Grefenstette found a set of values that significantly improved on-line performance compared to De Jong's values, but he could not improve off-line performance:

populations size $n = 30$,
crossover rate $p_c = 0.95$ and
mutation rate $p_m = 0.01$.

Note that these values bring as a windfall-profit a nice acceleration of the computational results — a population of 30 members is much quicker evaluated as one of 100 members.

Recall that Baker (1987) stated $O(n \cdot l)$ for the complexity of the GA; we get thus a linear reduction of computing time.

The perhaps most important study of parameter values was presented rather recently. In their paper, Schaffer, Caruana, Eshelman and Das (1989) claim to have accomplished *"the most thorough study to date of the influence of these control parameters* [Note : n, p_m, p_c as well as the number of crossover points] *control parameters on genetic on genetic search"*. Schaffer et al. (1989) considered the on-line performance of the functions of the De Jong (1975) test suite plus five new functions in an attempt to widen search space characteristics. These new functions included a FIR filter with about 50 million local minima, a 30-city TSP problem and a 64-node graph partitioning problem. Among the factors fixed for the experiment were Gray coding, the elitist strategy and Baker's SUS sampling algorithm.

Schaffer et al. (1989) used 10 replications each on the set of six population sizes, ten crossover rates, seven mutation rates and two crossover types. This results in 840 different settings, or 'cells' as Schaffer et al. (1989) called them, each being executed ten times. This factorial design was followed by an analysis of variance to identify and quantify the influences of these control parameters. Using a level of $\alpha = 0.05$, they constructed a 'best on-line pool', that is the subset of cells whose mean on-line performance was statistically indistinguishable from the best single cell in the set. We will focus here on their function-independent results, as the function that we are going to present later does not ressemble any of these functions. Schaffer et al. (1989) found that two-point crossover did *"no worse and sometimes better than one-point crossover.* [...] *At low population size, performance is very sensitive to mutation rate and less so to crossover rate.* [...] *As population size is increased, the sensitivity to mutation rate decreases and the best mutation rate to use also decreases. The generally inverse relation between population size and mutation probably reflects the fact that increasing either one increases exploration; they may be traded off while maintaining a more or less constant level of exploration.* [...] *High crossover is best at low populations, a broad range of crossover rates is tolerated at the middle population sizes and only low crossover rates are suggested for the high population sizes"*. Schaffer et al. (1989) finally advocated for

populations size n between $n = 20$ and $n = 30$,
crossover rate p_c between $p_c = 0.75$ and $p_c = 0.95$ and
mutation rate p_m between $p_m = 0.005$ and $p_m = 0.01$.

They concluded that *"the relative insensitivity to crossover rate suggests that these winning cells may be relying on a strategy consisting mainly of selection and mutation, that we might call naive evolution* (NE)".

Schaffer et al. (1989) also remarked that, while crossover rate could vary as discussed above, the product of population size, mutation rate and string length seemed to be constant in their "best on-line pool". They thus regressed on the logs of population size n, mutation rate p_m and bit string length l and obtained

$$\ln(n) + 0.9318 \ln(p_m) + 0.4535 \ln(l) = 0.5626 \ ,$$

but they failed to give any measure for the goodness-of-fit or parameter significance.

Schaffer et al. (1989) investigated only in on-line performance and acknowledged that off-line and generation-best performance will almost surely be different from their findings. Still, on-line might be most interesting figure as it takes into account *all* evaluations, the total computing cost of a genetic algorithm run.

3.7 Summary

In this chapter, we discussed more performance oriented aspects of genetic algorithms. Starting from the standard GA described earlier, we discussed the topics of

- representation where we focused on the aspects relevant for numerical optimisation,
- initialization where we mentioned hybrid systems that make use of heuristics equipped with domain knowledge,
- evaluation functions and constraints possibly attached to the problem to be optimised,
- the genetic algorithm operators reproduction, i.e. selection methods and the sampling algorithm by Baker (1987), crossover where we considered two-point crossover as well as Syswerda's (1989) uniform crossover, and mutation,
- parameter settings where we discussed the results found by De Jong (1975), Grefenstette (1986) and Schaffer et al. (1989).

We have hereby completed the presentation of genetic algorithms; after an introduction, some theoretical results and these different issues, we will now turn towards the problem domain.

Chapter 4

Some Elements of Financial Theory

Attacks on efficiency belong, of course, in the camp of the devil.
EUGENE F. FAMA (1991)

4.1 Introduction

In this chapter, we will first consider the problem of decision making under uncertainty and its representation by means of expected value and standard error of return due to the pioneering article of Markowitz (1952). Next, we will present the single-index model, a seminal simplification of portfolio analysis due to Sharpe (1964). Thereafter, we will introduce the capital asset pricing model (or CAP-M for short). We then discuss briefly the different forms of the efficient markets hypothesis. A summary will conclude the chapter.

4.2 Decision under Uncertainty and Portfolio Analysis

If we want to understand how asset prices are fixed on financial markets, we have to see how economic agents rank different assets promising different future payments. We thus consider the decision making among uncertain alternatives (Kreps 1990, Göppl 1990). The theory of decision under uncertainty is based on five axioms. We consider the set $\mathcal{X}$ of outcomes for assets, i.e. payments that the economic agent will receive at the end of the period. We write $\tilde{x}$, $\tilde{y}$ and $\tilde{z}$ for elements of $\mathcal{X}$. The tilde shows that these events are uncertain in the sense that the economic agents have probability distributions for these outcomes. The preference is noted as $\succ$, the indifference as $\sim$ and $\succeq$ stands for 'preferred or indifferent'.

We want to ensure that the investor is able to compare all elements of the set $\mathcal{X}$:

Axiom 4.1 $\forall\, \tilde{x}, \tilde{y} \in \mathcal{X}^2$ *either* $\tilde{x} \succ \tilde{y}$ *or* $\tilde{y} \succ \tilde{x}$ *or* $\tilde{x} \sim \tilde{y}$.

The property of transitivity is also needed:

Axiom 4.2 $\forall\, \tilde{x}, \tilde{y}, \tilde{z} \in \mathcal{X}^3$

$$\begin{aligned} \tilde{x} \succeq \tilde{y} \text{ and } \tilde{y} \succ \tilde{z} &\implies \tilde{x} \succ \tilde{z} \\ \tilde{x} \succ \tilde{y} \text{ and } \tilde{y} \succeq \tilde{z} &\implies \tilde{x} \succ \tilde{z} \\ \tilde{x} \sim \tilde{y} \text{ and } \tilde{y} \sim \tilde{z} &\implies \tilde{x} \sim \tilde{z} \,. \end{aligned}$$

We need some more notation for the next axiom. Consider a lottery G with two possible payments $\tilde{x}$ and $\tilde{y}$ where $\tilde{x}$ has an outcome probability of α and $\tilde{y}$ hence of $1-\alpha$. We will write $G(\tilde{x}, \tilde{y} : \alpha)$ for this lottery. Thus we can formulate the strict independence:

Axiom 4.3 $\forall\, \tilde{x}, \tilde{y}, \tilde{z} \in \mathcal{X}^3\ \forall\, \alpha \in [0,1] \quad \tilde{x} \sim \tilde{y} \implies G(\tilde{x}, \tilde{z} : \alpha) \sim G(\tilde{y}, \tilde{z} : \alpha)$,

if the investor cannot make up his mind between two assets, then there is absolutely no asset that does influence his judgement: he remains indifferent. Next, we want measurability, that is the existence of a probability such that the investor becomes indifferent:

Axiom 4.4 $\forall\, \tilde{x}, \tilde{y}, \tilde{z} \in \mathcal{X}^3 \quad \tilde{x} \succ \tilde{y} \succeq \tilde{z}$ *or* $\tilde{x} \succeq \tilde{y} \succ \tilde{z} \implies \exists\, \alpha \in [0,1] \quad \tilde{y} \sim G(\tilde{x}, \tilde{z} : \alpha)$.

Finally, we want a ranking — the probabilities are ordinal measures of the preferences among the assets.

Axiom 4.5 $\forall\, \tilde{x}, \tilde{y}, \tilde{z}, \tilde{u} \in \mathcal{X}^4 \quad \exists\, \alpha_1, \alpha_2 \in [0,1]$

$$\left.\begin{array}{r} \tilde{x} \succeq \tilde{y} \succ \tilde{z} \\ \tilde{x} \succeq \tilde{u} \succ \tilde{z} \\ \tilde{y} \sim G(\tilde{x}, \tilde{z} : \alpha_1) \\ \tilde{u} \sim G(\tilde{x}, \tilde{z} : \alpha_2) \end{array}\right\} \quad \begin{array}{l} \alpha_1 > \alpha_2 \iff \tilde{y} \succ \tilde{u} \\ \alpha_1 = \alpha_2 \iff \tilde{y} \sim \tilde{u} \end{array}$$

Fama and Miller (1972) showed that the concept of expected utility developed by von Neumann and Morgenstern (see Kreps 1990, chapter 3) is equivalent to this axiomatic system. We get thus a utility function U that is defined over the set $\mathcal{X}$ and that maps to the real line $\mathbb{R}$. This utility function is in line with the pre-order of alternatives:

$$U(\tilde{x}) > U(\tilde{y}) \iff \tilde{x} \succ \tilde{y} \quad \text{and} \quad U(\tilde{x}) = U(\tilde{y}) \iff \tilde{x} \sim \tilde{y} \,.$$

Especially, we can calculate the expected values of the lotteries G:

$$U(G(\tilde{x}, \tilde{y} : \alpha)) = \alpha U(\tilde{x}) + (1-\alpha) U(\tilde{y}) \,.$$

One usually wants that the economic agents are risk-averse, i.e. that a certain outcome is always preferred to a lottery having an expected value equal to the certain value:

$$U(E(.)) > E(U(.)) \,.$$

We will now pass over to the (μ, σ)-concept of Markowitz (1952). Instead of considering a payment $\tilde{x} \in \mathcal{X}$ for an asset, we will henceforth consider its return over the period:

$$\tilde{r}_{x,s} = \frac{\tilde{x}_s - w_0}{w_0} \,.$$

The index s stands for the dependence of the outcome $\tilde{x}$ from a certain 'state of the world' s; w_0 stands for the asset's price at the beginning of the period. With axioms 4 and 5, we can use the probabilities to rank different alternatives from $\mathcal{X}$. These alternatives give an uncertain return over the period. As it is difficult to compare several distributions which could as well follow different laws, we will assume that the returns are normally distributed.

One tuple of (μ, σ^2)–values describes completely one normal law. We can now consider the utility maximization for pairs of $(\mu_x, \sigma_x^2)\ \forall\, x \in \mathcal{X}$ which is equivalent to the maximization of expected utility and to the axiomatic concept of decision under uncertainty. Note that this is a generalization that enables us to consider every possible (risk-averse) investor without having to regard his utility function explicitly — all we need are the (μ_x, σ_x^2) values for the assets.

Hence, we calculate expected value and variance of the return of an asset $\tilde{x} \in \mathcal{X}$ where the return depends on the realisation of a 'state of the world' from the set of possible states $s = 1, \ldots, S$ with probabilities p_s as

$$\mu_x = E(\tilde{r}_x) = \sum_{s=1}^{S} p_s \tilde{r}_{x,s} \quad \text{and} \tag{4.1}$$

$$\sigma_x^2 = \sum_{s=1}^{S} p_s (\tilde{r}_{x,s} - \mu_x)^2 \ . \tag{4.2}$$

Note that we will for notational convenience drop the indices x or i when we refer to μ or σ, but this is precisely what we mean: the expected value of the return and the return standard error (resp. variance for σ^2) of an asset x or i. Henceforth, we will need the covariance between two assets

$$\sigma_{ij} = \operatorname{cov}(\tilde{r}_i, \tilde{r}_j) = \sum_{s=1}^{S} p_s (\tilde{r}_i - \mu_i)(\tilde{r}_j - \mu_j) \tag{4.3}$$

as well as their correlation coefficient

$$\rho_{ij} = \operatorname{corr}(\tilde{r}_i, \tilde{r}_j) = \frac{\sigma_{ij}}{\sigma_i \sigma_j} \ . \tag{4.4}$$

The next step is to look at combinations of assets. We suppose that there are N assets in the set $\mathcal{X}$. We denote the fraction of the value of the portfolio that is made up by asset i by x_i where of course $\sum_i x_i = 1$ and $x_i \geq 0\ \forall\, i$. We obtain for the portfolio return

$$\mu_P = \tilde{r}_P = \sum_{i=1}^{N} x_i \tilde{r}_i \tag{4.5}$$

where the tilde signals that this return is uncertain and for the portfolio variance, we obtain

$$\sigma_P^2 = \sum_{i=1}^{N} x_i^2 \sigma_i^2 + \sum_{i=1}^{N} \sum_{\substack{j=1 \\ j \neq i}}^{N} x_i x_j \sigma_{ij} = \sum_{i=1}^{N} \sum_{j=1}^{N} x_i x_j \sigma_{ij} \ . \tag{4.6}$$

Equation (4.6) shows that the portfolio risk depends upon the factors asset weights (x_i, x_j), the asset variances (σ_i^2) and the asset covariances (σ_{ij}). Suppose that we set the weights to $x_i = 1/N\ \forall\, i = 1, \ldots, N$. Consider then first the case where all covariances are zero. The portfolio variance equals thus

$$\sigma_P^2 = \frac{1}{N^2} \sum_{i=1}^{N} \sigma_i^2$$

and for increasing values of N, we obtain an investment with decreasing risk. Still, it is not easy to find stocks that have zero covariance. So let us suppose that the covariances do not cancel out and that we invest in equal parts. Equation (4.6) becomes

$$\sigma_P^2 = \frac{1}{N} \sum_{i=1}^{N} \frac{\sigma_i^2}{N} + \frac{N-1}{N} \sum_{i=1}^{N} \sum_{\substack{j=1 \\ j \neq i}}^{N} \frac{\sigma_{ij}}{N(N-1)} = \frac{1}{N} \overline{\sigma_j^2} + \frac{N-1}{N} \overline{\sigma_{ij}} \ ,$$

where the bars in the last part of the equation denote average values. So for increasing values of N, the part of each asset's variance in the portfolio vanishes and the portfolio variance approaches the average asset covariance.

We will now consider the case of two assets that we denote by indices A and B and regard the impact of different covariance values. Using the identity $x_a + x_b = 1$, and the definition of the correlation coefficient in (4.4), equation (4.6) simplifies to

$$\sigma_P^2 = x_A^2 \sigma_A^2 + (1 - x_A)\sigma_B^2 + 2x_A(1 - x_A)\rho_{AB}\sigma_A\sigma_B \ . \tag{4.7}$$

We will start by supposing that the two assets are perfectly correlated: $\rho_{AB} = 1$. Equation (4.7) becomes thus

$$\sigma_P^2 = x_A^2 \sigma_A^2 + (1 - x_A)\sigma_B^2 + 2x_A(1 - x_A)\sigma_A\sigma_B = (x_A\sigma_A + (1 - x_A)\sigma_B)^2$$

so that σ_P, the portfolio return standard error, is a linear function of the two asset standard errors. So in (μ, σ)–space, the set of all possible portfolio made from two asset whose that are perfectly correlated is a straight line.

Next we suppose that the two assets have perfect negative correlation: $\rho_{AB} = -1$. This yields $\sigma_P^2 = (x_A\sigma_A - (1 - x_A)\sigma_B)^2$ and hence the standard error expression

$$\sigma_P = |x_A\sigma_A - (1 - x_A)\sigma_B|$$

is also a linear combination of the two standard errors. If we solve $x_A\sigma_A - (1 - x_A)\sigma_B = 0$, we get

$$x_A = \frac{\sigma_A}{\sigma_A + \sigma_B} \ ,$$

the share of the first asset in a riskless portfolio constructed from two perfectly negatively correlated assets. This is a general solution; unfortunately one rarely finds assets with such a correlation.

Finally, we suppose that the correlation is somewhere in between these two values: $|\rho_{AB}| < 1$. We obtain for the portfolio standard error

$$\sigma_P = \left(x_A^2\sigma_A^2 + (1 - x_A)^2\sigma_B^2 + 2x_A(1 - x_A)\sigma_A\sigma_B\rho_{AB}\right)^{1/2} \ .$$

We get the minimal-variance portfolio by deriving this with respect to x_A:

$$\frac{\partial \sigma_P}{\partial x_A} = \frac{2x_A\sigma_A^2 - 2\sigma_B^2 + 2x_A\sigma_B^2 + 2\sigma_A\sigma_B\rho_{AB} - 4x_A\sigma_A\sigma_B\rho_{AB}}{2\left(x_A^2\sigma_A^2 + (1 - x_A)^2\sigma_B^2 + 2x_A(1 - x_A)\sigma_A\sigma_B\rho_{AB}\right)^{1/2}} \equiv 0$$

which we solve for x_A:

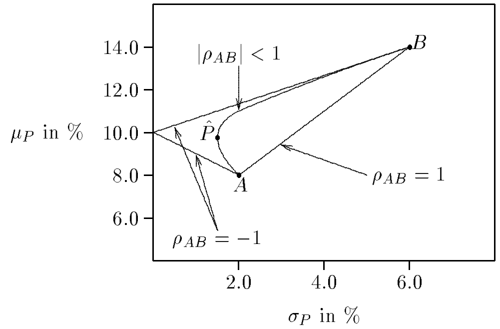
$$x_A = \frac{\sigma_b^2 - \sigma_A\sigma_B\rho_{AB}}{\sigma_A^2 + \sigma_B^2 - 2\sigma_A\sigma_B\rho_{AB}} \ .$$

If $\rho_{AB} = 0$, the last expressions simplifies to

$$x_A = \frac{\sigma_b^2}{\sigma_A^2 + \sigma_B^2} \ .$$

We summarize these different cases in figure 4.1.

Figure 4.1: Expected value and standard error of the return for a portfolio of two assets for three different correlation coefficients



Source: Similar figures can be found in *any* modern finance textbook, e.g. Elton and Gruber (1984), page 53.
Note: The arrows do not show single points that have the indicated values for ρ_{AB}; the whole line resp. curve corresponds to one value.

After having outlined the principal possibilities to combine assets to form a portfolio, we have to see if we can guide the selection of the various portfolios. Let us assume that the correlation coefficient lies somewhere between -1 and 1. The set of realisable portfolios to be formed from assets A and B is then a curved line as in figure 4.1. Which (μ, σ)–combination will an investor choose? Consider the the portfolio represented by the point $\hat{P}$. It clearly gives more return with less risk than the portfolio consisting of asset A only. So A is dominated by $\hat{P}$ — as well as all the other portfolios between A and $\hat{P}$. What is left over is the set of portfolios that lie between $\hat{P}$ and B on a curve segment that is known as the *efficient frontier*. Portfolios from the efficient frontier are efficient in the sense that you cannot reduce σ any more for a given μ or, inversely, get a higher μ for a given σ. $\hat{P}$ represents the *minimal variance portfolio*. But this is not necessarily the portfolio chosen by any investor. It clearly is the most attractive opportunity for *extremely* risk-averse investors whereas B attracts the risk-loving investors. All other investors choose a combination somewhere between these two solutions.

We will now introduce a third asset: a riskless bond that reports r_F %. We suppose that all investors can lend and borrow at this rate. As its return is supposed to be certain, its standard error is zero and hence the point lies on the μ–axis in (μ, σ)–space. The investor can now combine this riskless asset with a risky asset, say asset B from above. He will invest $0 \le x_B \le 1$ parts in the risky asset and hence $1 - x_B$ in the riskless bond. Using equations

(4.5) and (4.6), we get

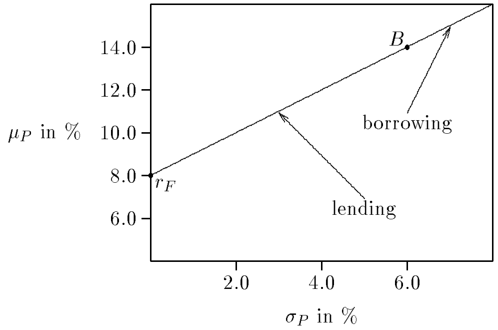
$$\begin{aligned} \mu_P &= x_B\mu_B + (1 - x_B)r_F \quad \text{and} \\ \sigma_P &= \left(x_B^2\sigma_B^2 + (1 - x_B)^2\, 0 + 2x_B(1 - x_B)\sigma_B\, 0\, \rho_{FB}\right)^{1/2} \\ &= x_B\sigma_B \quad . \end{aligned}$$

So we obtain $x_B = \sigma_P/\sigma_B$ which we substitute in the μ_P-equation:

$$\mu_P = \frac{\sigma_P}{\sigma_B}\mu_B + \left(1 - \frac{\sigma_P}{\sigma_B}\right) r_F = r_F + \frac{\mu_B - r_F}{\sigma_B}\sigma_P \quad .$$

So all combinations of the riskless asset and the risky asset form on a straight line in (μ, σ)-space, see figure 4.2. This is also a perfectly general result.

Figure 4.2: Expected value and standard error of the return for a portfolio of a risky and a riskless asset.



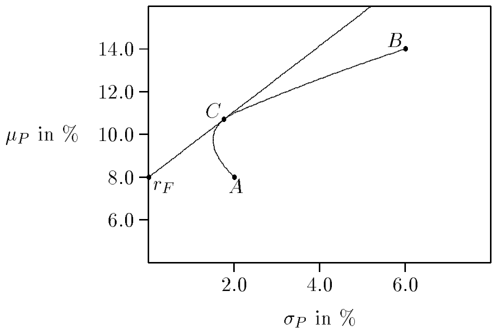
Source: Elton and Gruber (1984), page 64.
Note: Left of point B, we have combinations of lending at rate r_F and asset B; right of B we have combinations of borrowing at the same rate and of portfolio B.

We could replace the asset B by a portfolio — for this analysis, there is strictly no difference between assets and portfolios as the only characteristics that we consider are uncertain payments and the resulting expected value and standard error, respectively.

As well, we can combine the riskless asset with any portfolio from the efficient frontier. Which would be the optimal choice? We are interested in the steepest slope of the straight line in (μ, σ)–space as this guarantees the biggest possible reward (μ) for any risk (σ) that we cover. The outermost point on the efficient frontier that could form a straight line with the point $(0, r_F)$ is the tangency point, see figure 4.3 where point C is the tangency point. This line through the point C is called the *efficient line*.

The portfolio that is represented by point C is an interesting one. Every investor facing riskless borrowing and lending at rate r_F and the efficient frontier between A and B will hold this same portfolio of assets. If the investor is rather risk-averse, he will place some of its money at the rate r_F and some money in the portfolio C, his resulting portfolio is thus

Figure 4.3: Expected value and standard error of the efficient line.



Source: Elton and Gruber (1984), page 64.

somewhere on the line between $(0, r_F)$ and C. Inversely, a rather risk-loving investor will borrow at r_F to be able to buy even more of the portfolio C; his portfolio will lie somewhere to the right hand side of C on the ray passing through C. So independently of his utility function and thus risk preference, every investor holds the same portfolio C of risky assets. This result is called the *separation theorem*. Elton and Gruber (1984, chapter 4) provide a simple method to calculate the efficient frontier. One searches the outermost tangent to the curve from A to B, i.e. the ray with maximum slope. Instead of using Lagrangian multipliers, they substitute the constraint into the objective function and take N derivatives with respect to x_i. This $N \cdot N$–system can be solved directly giving the N values, that, together with equations (4.5) and (4.6) yield the coordinates of the tangential point (e.g. of C in figure 4.3).

4.3 The Single-Index Model of Sharpe (1964)

In a seminal paper, Sharpe (1964) showed a way to simplify portfolio analysis. Consider once again equations (4.5) and (4.6). Suppose now that you want to calculate the efficient frontier for, say, 500 stocks. So what you need as input for these two equations are 500 expected returns, 500 variances and $1/2 \cdot 500 \cdot 499 = 124\,750$ covariances! Moreover, these values will be estimated; so you have to ensure the quality of more than 125 000 estimates. But empirically, all stocks seem to follow “the market” where this usually means a market index covering a broad range of shares. This suggests that the share return and market return are correlated:

$$\tilde{r}_i = \alpha_i + \beta_i \tilde{r}_M + e_i$$

where	$\tilde{r}_i$	return of stock i; random variable
	α_i	non-market dependent component of r_i; constant
	β_i	expected change in $\tilde{r}_i$ given a change in $\tilde{r}_M$; constant
	$\tilde{r}_M$	return of the market or market index; random variable with $\text{var}(\tilde{r}_M) = \sigma_M^2$

e_i	residual component with
$E(e_i) = 0$	$\forall\, i = 1, \ldots, N$ by construction
$\text{var}(e_i) = E(e_i)^2 = \sigma_{ei}^2$	$\forall\, i = 1, \ldots, N$ by definition
$\text{cov}(\tilde{r}_M, e_i) = 0$	$\forall\, i = 1, \ldots, N$, by assumption
$E(e_i e_j) = 0$	$\forall\, i = 1, \ldots, N\ j = 1, \ldots, N\ j \neq i$, by assumption.

So the expected return on a security is given by

$$\mu_i = E(\tilde{r}_i) = E\left(\alpha_i + \beta_i \tilde{r}_M + e_i\right) \ .$$

As $E(a+b) = E(a) + E(b) = a + E(b)$ where the last equality holds for any constant a, we get for the stock return in the single-index model:

$$\begin{aligned} \mu_i &= E(\tilde{r}_i) = \alpha_i + \beta_i E(\tilde{r}_M) \\ &= \alpha_i + \beta_i \mu_M \end{aligned} \qquad (4.8)$$

because e_i cancels out as its expectation is zero. The variance of a stock return is thus

$$\sigma_i^2 = E\left[(\alpha_i + \beta_i \tilde{r}_M + e_i) - (\alpha_i + \beta_i \mu_M)\right]^2 = E\left[\beta_i(\tilde{r}_M - \mu_M) + e_i\right]^2$$

where the middle term vanishes after squaring because of the $\text{cov}(e_i, \tilde{r}_M) = 0$ assumption so that

$$\begin{aligned} \sigma_i^2 &= \beta_i^2 E(\tilde{r}_M - \mu_M)^2 + E(e_i)^2 \\ &= \beta_i^2 \sigma_M^2 + \sigma_{ei}^2 \ . \end{aligned} \qquad (4.9)$$

For the covariance

$$\sigma_{ij} = E[(\tilde{r}_i - \mu_i)(\tilde{r}_j - \mu_j)] \ ,$$

substitution yields

$$\sigma_{ij} = E\Big[[(\alpha_i + \beta_i \tilde{r}_M + e_i) - (\alpha_i + \beta_i \mu_M)][(\alpha_j + \beta_j \tilde{r}_M + e_j) - (\alpha_j + \beta_j \mu_M)]\Big]$$

and rearranging gives

$$\sigma_{ij} = E[(\beta_i(\tilde{r}_M - \mu_M) + e_i)(\beta_j(\tilde{r}_M - \mu_M) + e_j)]$$

and after multiplication, all terms with an e_i disappear so that

$$\sigma_{ij} = \beta_i \beta_j \sigma_M^2 \ . \qquad (4.10)$$

Recall that the return of a portfolio is given by

$$\mu_P = \sum_{i=1}^{N} x_i \mu_i = \sum_{i=1}^{N} x_i E(\tilde{r}_i)$$

where we obtain by substituting from (4.8)

$$\mu_P = \sum_{i=1}^{N} x_i \alpha_i + \sum_{i=1}^{N} x_i \beta_i \mu_M \ . \qquad (4.11)$$

Similarly, we have portfolio variance as

$$\sigma_P^2 = \sum_{i=1}^N x_i^2\sigma_i^2 + \sum_{i=1}^N \sum_{\substack{j=1\\j\neq i}}^N x_i x_j \sigma_{ij}$$

where we substitute σ_i^2 from (4.9) and σ_{ij} from (4.10); this yields

$$\sigma_P^2 = \sum_{i=1}^N x_i^2\beta_i^2\sigma_M^2 + \sum_{i=1}^N \sum_{\substack{j=1\\j\neq i}}^N x_i x_j \beta_i\beta_j\sigma_M^2 + \sum_{i=1}^N x_i^2\sigma_{ei}^2 \,. \tag{4.12}$$

We can now estimate portfolio return and portfolio risk provided estimates for α_i, β_i, σ_{ei}^2 for each stock as well as estimates for μ_M and σ_M^2. This totals $3N+2$ and compares favourably to the $N(N-1)/2$ estimates that we needed beforehand.

We will analyze this single-index model further. Let us define

$$\alpha_P = \sum_{i=1}^N x_i\alpha_i \text{ and } \beta_P = \sum_{i=1}^N x_i\beta_i$$

so that we can write equation (4.11) as

$$\mu_P = \alpha_P + \beta_P\mu_M \,.$$

So if we take the market portfolio as the portfolio P, it follows that we obtain $\alpha_P = 0$ and $\beta_P = 1$. Equally, we can rewrite equation (4.10) as

$$\begin{aligned}\sigma_P &= \sum_{i=1}^N\sum_{j=1}^N \beta_i\beta_j\sigma_M^2 + \sum_{i=1}^N x_i^2\sigma_{ei}^2 \\ &= \sum_{i=1}^N x_i\beta_i \sum_{j=1}^N \beta_j x_j\sigma_M^2 + \sum_{i=1}^N x_i^2\sigma_{ei}^2 \\ &= \beta_P^2\sigma_M^2 + \sum_{i=1}^N x_i^2\sigma_{ei}^2 \,.\end{aligned}$$

Let us assume as on page 31 that the investor places equal amounts $1/N$ on every stock; the last equation becomes thus

$$\sigma_P = \beta_P^2\sigma_M^2 + \frac{1}{N}\sum_{i=1}^N \frac{1}{N}\sigma_{ei}^2 \,.$$

The last term equals thus the average residual risk divided by N so that this *naive diversification strategy* of investing equal parts in a growing number of stocks diminishes the residual risk. Assuming that residual risk approaches zero, we write portfolio risk as

$$\sigma_P = \left[\beta_P^2\sigma_M^2\right]^{1/2} = \beta_P\sigma_M = \sigma_M\sum_{i=1}^N x_i\beta_i \,; \tag{4.13}$$

this is often referred to as *market risk*, as *non-diversifiable risk* or as *systematic risk* to express that this risk depends on the market as a whole. Consequently, σ_{ei}, which can be diversified, is called *non-market risk*, *diversifiable risk* or *unsystematic risk*. Equation (4.13) shows clearly that the market risk in a portfolio depends on the β_i–factors of its shares. Estimation of β is thus an important problem in practical portfolio analysis matters. We will not discuss β–estimation, see Elton and Gruber (1984, chapter 5) for a discussion of the subject.

4.4 The Capital Asset Pricing Model

The *capital asset pricing model*, or CAP-M for short, was developed independently by Sharpe (1964), Lintner (1965), and Mossin (1966). It is based on the set of assumptions presented in table 4.1:

Table 4.1: The assumptions for the standard capital asset pricing model

Absence of transaction cost	The economic agents can buy and sell without cost.
Divisible goods	All traded asset are infinitely divisible; investors can take any position regardless of size and wealth.
No personal tax	The investors are indifferent to the form (i.e. dividends or capital gains) in which return on investment is received.
Perfect competition	An individual economic agents cannot influence prices by his actions of buying and selling; though as a whole, all investors determine prices.
Portfolio theory framework	Investors are expected to make decisions solely in terms of expected values and standard deviations of stock returns.
Unlimited short sales	The economic agents can sell short any quantity of stock.
Unlimited lending and borrowing	The economic agents can lend and borrow any amount at the interest rate r_F; these are riskless investments ($\sigma_F = 0$).
Period length	All investors plan in terms of expected value and standard error over a period of the same length.
Identical expectations	All investors are supposed to have the same expectations for the necessary input (expected returns, standard errors and covariances).
Marketability	All assets, including human capital, can be bought and sold on the market.

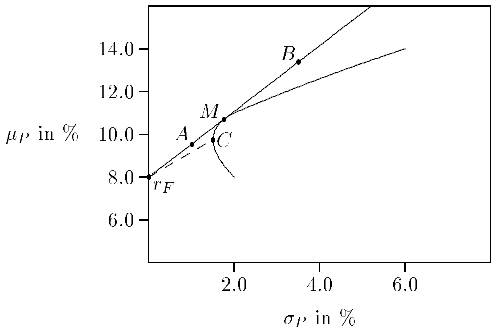
Source: Elton and Gruber (1984), pages 276 and 277.

As all investors have identical expectations *by assumption*, there is only one efficient frontier, that is one curved line with the non-dominated portfolios. At as there is only riskless lending and borrowing rate, all investors face the same tangency point between the ray starting at this rate and the efficient frontier.

If we define v_i as the market value of all shares of company i, their part x_i in the total value of all assets is given by $x_i = v_i/\sum_i v_i$. Evidently, we have $\sum_i x_i = 1$, but the remaining question is: how does the equilibrium market portfolio look like — and how do investors divide their wealth among the different assets. Each investor has two basic choices. He either invests in risky assets, taking a (non-dominated) portfolio somewhere from the efficient frontier or he invests in the riskless asset at the interest rate r_F. The investor tries to maximize his utility with respect to μ and σ.

Consider now figure 4.4. Let us assume that the investor has chosen the portfolio represented by the point C. He can now reduce his risk by additionally placing money at the rate r_F; he thus obtains portfolios constructed by choosing combinations of his asset portfolio C and the riskless bond as depicted by the dashed line.

Figure 4.4: Deriving the standard capital asset pricing model.



Source: Göppl (1990), page 94; Elton and Gruber (1984), page 279.

One realises immediately that the investor can maximize his utility by switching from the portfolio C to another portfolio on the efficient frontier — the portfolio M which is the tangency point. In figure 4.4, one sees that the ray starting at $(0, r_F)$ and passing through the point M dominates all other combinations of μ and σ. This ray is called the *capital market line* or CML for short. Following the ninth assumption, all investors have homogeneous expectations. It follows that the CML is the new set of efficient (in the sense of non-dominated) portfolios. The point M is thus the market portfolio covering all tradeable assets. As the CML is a linear combination between the riskless asset and the market portfolio, each investor realises the market portfolio in combination with the riskless asset. This means that each investors divides his wealth among the assets as they are distributed in the market portfolio; the proportions x_i are the same for all investors and the same as in the market portfolio M.

One can equally well derive the CAP-M and the CML by maximizing θ, the slope of the ray starting at $(0, r_F)$; see for example Elton and Gruber (1984), pages 285–287. This yields the analytical form of the capital market line:

$$\tilde{r}_e = r_F + \frac{\mu_M - r_F}{\sigma_M}\sigma_e \ . \tag{4.14}$$

This is a linear risk-return relationship; the more return an investors wants to earn, the more risk he has to cover. For no risk at all, we obtain r_F; this is the reward for postponing today's expenditure until the end of the period or inversely the price of consuming today what will be payed at the end of the period. Thus, equation (4.14) can be viewed as

$$\text{(Expected Return)} = \text{(Price of time)} + \text{(Price of risk)} \times \text{(Amount of risk)}$$

where $(\mu_m - r_F)/\sigma_M$ denotes the price of risk — the return you get for another unit of risk — and σ_e denotes the amount of risk in the (efficient) portfolio on the CML.

What is the relationship between risk and return for a single stock? Consider an inefficient share that, say, does not reward sufficiently its risk. We would plot it thus below the capital market line. Consider then a portfolio of the market portfolio M and this inefficient portfolio. We note x_M the part of the market portfolio in this new portfolio, x_i the part of the inefficient portfolio and write with index P the return and risk of the new portfolio. We can easily express μ_P and σ_P by substituting into equations (4.5) and (4.6); this yields

$$\begin{aligned} \mu_P &= x_i\mu_i + x_M\mu_M = x_i\mu_i + (1 - x_i)\mu_M \\ \sigma_P &= \left(x_i^2\sigma_i^2 + (1 - x_i)^2\sigma_M^2 + 2x_i(1 - x_i)\sigma_{i,M}\right)^{1/2} \ . \end{aligned}$$

We now consider directly the influence of a change in x_i for the portfolio:

$$\begin{aligned} \frac{\partial\mu_P}{\partial x_i} &= \mu_i - \mu_M \\ \frac{\partial\sigma_P}{\partial x_i} &= \frac{x_i\left(\sigma_i^2 + \sigma_M^2 - 2\sigma_{i,M}\right) + \sigma_{i,M} - \sigma_M^2}{\sigma_P} \ . \end{aligned}$$

The slope of the curve that describes the portfolios made up from the inefficient portfolio i and the market portfolio M is given by

$$\frac{\partial\mu_P}{\partial\sigma_P} = \frac{\partial\mu_P}{\partial x_i}\frac{\partial x_i}{\partial\sigma_P} = \frac{(\mu_i - \mu_M)\sigma_P}{x_i\left(\sigma_i^2 + \sigma_m^2 - 2\sigma_{i,M}\right) + \sigma_{i,M} - \sigma_M^2} \ .$$

For the market portfolio M, we get $x_i = 0$ and thus $\sigma_P = \sigma_M$; this yields

$$\left.\frac{\partial\mu_P}{\partial\sigma_P}\right|_{x_i=0} = \frac{(\mu_i - \mu_M)\sigma_M}{\sigma_{i,M} - \sigma_M^2} \ .$$

We know as well the slope of the CML, i.e. the slope of the ray passing through M so that we can set

$$\frac{\mu_m - r_F}{\sigma_M} = \frac{(\mu_i - \mu_M)\sigma_M}{\sigma_{i,M} - \sigma_M^2} \ .$$

We can solve this last equation for the "additional return" $\mu_i - r_F$ that rewards covering the risk in stock i:

$$\mu_i - r_F = \frac{\mu_M - r_F}{\sigma_M^2}\sigma_{i,M} \ . \tag{4.15}$$

This result is valid for all stocks and all portfolios of stocks.

Hence, in $(\mu_i, \sigma_{i,M})$-space, this equation must cover the riskless asset reporting r_F as well as the market portfolio. We obtain a ray starting at $(0, r_F)$ and passing through (μ_M, σ_M^2). This ray is called the *security market line*; we depicted it in figure 4.5. We can express this relation by using the beta-coefficient used earlier. If we set

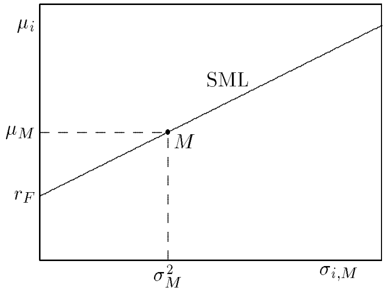
$$\frac{\sigma_{i,M}}{\sigma_M^2} \equiv \beta_i \ ,$$

we get, changing slightly equation (4.15),

$$\mu_i = r_F + \beta_i(\mu_M - r_F) \ ,$$

the equation of the security market line. There are three possible interpretations for β:

Figure 4.5: The security market line



Source: Göppl (1990), page 102.

- β is a stock-specific coefficient that weights the risk premium of the market,
- β gives the sensitivity of a stock return to changes in "additional market return" and
- β is a regression parameter for estimating the relation between the return on an individual stock and the market return.

Following this second interpretation, we can distinguish between *aggressive stocks* with $\beta > 1$ that lie to the right of point M in figure 4.5 and *defensive stocks* with $\beta < 1$ lying to the left of this point that represents the market portfolio.

4.5 On Market Efficiency

The concept of indexation, that we will present shortly, is closely related to the notion of market efficiency. We will therefore discuss the latter now. The general references are Fama (1970), Fama (1976) and Fama (1991) along with a many textbooks on financial economics.

Fama (1991) has recently provided a review. He there defines the *market efficiency hypothesis* as *"the simple statement that security prices fully reflect all available information"*. This is equivalent to the more formal definition that *"if capital markets are efficient, then purchase or sale of any security at the prevailing market price is a zero-NPV* [Note: net present value] *transaction"* (Brealey and Myers). Fama states as well a weaker definition due to Jensen that *"prices reflect information to the point where the marginal benefits of acting on information do not exceed the marginal costs"*.

Fama (1991) then renames the three common labels for market efficiency. He speaks of *tests for return predictability* instead of weak-form tests to refer to tests to answer the question whether, and if, how well, past returns do predict future returns. Instead of semi-strong-form tests, he now uses the term *event studies* as this reflects better that price changes are supposed to be caused by public announcements — events. Finally, for the strong-form tests

of whether some investors have information not reflected in market prices, he uses *tests for private information*. We will now discuss these three different forms of the market efficiency hypothesis.

For answers to the question of return predictability, Fama (1991) further divides this into different parts. For short horizons, he concludes that *"with the CRSP* [Note: Center for Research in Security Prices] *daily data back to 1962, recent research is able to show confidently that daily and weekly returns are predictable from past returns"*. For longer horizons, the question is less clear as irrational bubbles are indistinguishable from rational time-varying expected returns. For other forecasting variables, there is recent evidence on the predictability of returns for short and long horizons. Among the variables used as input are dividend yields, earnings/prices-ratios, default spreads of low- over high-grade bond yields, term spreads and the level of short rates. There is, according to Fama (1991), more to learn about the links between expected returns and macro-variables.

We should also note that there exceptions — the "anomalies" in financial economics lingo. The most often quoted work is Banz (1981) who reported the *size effect*: small stocks tend to give a higher return. This is sometimes explained with a higher risk, a higher variance in these stocks, so that one is again in line with the theory. There are more anomalies, among them are the *weekend effect* and the *year end effect* that both state excess returns. Work on these anomalies has played a major part in emprirical financial economics in the last years; we are now quite comfortable with the Fama (1991) generally advocating for efficiency in spite of these anomalies.

Event studies are, for Fama, the cleanest evidence on efficiency — and its supportive evidence with few exceptions. Changes in dividends accord for a price change of the same sign, stock issues are considered as bad news, takeovers provide gains for the shareholders of the victim; and on average, stock prices seem to adjust within a day.

Finally, for private information, Fama (1991) states the early results that found that mutual-fund managers were able to beat the efficient line (constructed from the risk-free rate through the S&P 500 portfolio) by about 1% per year as well as recent work that advocates for the contrary. He notes the sensibility of this kind of evaluation to methodology, especially when the one-factor model is used. A recent three-factor model analysis to evaluate mutual-fund performance found an abnormal return of -1.1% per year. Concluding, Fama (1991) states that the one-factor model has problems explaining the cross-section of expected stock returns (e.g. size anomaly, weak relation between average return and β) and as multi-factor models seem to do a better job on expected returns, the multi-factor performance evaluation methods and their negative conclusions about the access of investment managers to private information are more reliable.

Fama (1991) then writes what now serves as a splendid connection to the next chapter on indexation: *"Since we are reviewing studies of performance evaluation, it is well to point out here that the efficient-markets literature is a premier case where academic research has affected real-word practice. Before the work on efficiency, the presumption was that private information is plentiful among investment managers. The efficiency research put forth the challenge that private information is rare. One result is the rise of passive investment strategies that simply buy and hold diversified portfolios (e.g. the many S&P 500 funds). Professional managers who follow passive strategies (and charge low fees) were unheard of in 1960; they are now an important part of the investment-management industry"*.

4.6 Summary

We have tried in this chapter to present the key elements of financial theory that are necessary for un understanding of the concept of indexation (or passive management as it is equally called). We have discussed

- the axiomatic base of decision under uncertainty and the (μ, σ)-analysis that is equivalent to it,
- the single-index model of Sharpe (1964) and the β-factors,
- the capital asset pricing model as an equilibrium model for financial assets and
- the three notions of market efficiency.

Chapter 5

On Indexation

If you can't beat them, join them!
HARRY M. MARKOWITZ (1990)

5.1 Introduction

This chapter gives an overview over *indexation*, or *passive management* as its equally called. For an introduction the subject, we will first present some arguments in favour of indexation taken from Bishop (1990a). Thereafter, we will consider the methods used for index fund construction — first as described by Liesching and Manchanda (1990), second as described by Meade and Salkin (1989). Finally, we will discuss fund maintenance issues raised by Meade and Salkin (1990).

5.2 Introducing Indexation (Bishop 1990b)

Bishop estimates that in the US between 20% - 25% of all pension fund assets are managed on an index basis; Markowitz (1990) gives a 250 billion $ estimate for the indexation volume worldwide. So what it is about? Bishop (1990b) defines *"the 'index fund' is a portfolio of securities which has been specifically designed to represent the characteristics and attributes of a chosen target index"*. There are several methods known that try to achieve the design; we will discuss them when we consider the work of Liesching and Manchanda (1990) and Meade and Salkin (1989, 1990) later in this chapter. Next Bishop (1990b) presents an informal definition for the *tracking error*, that is the difference between the index tracking portfolio and the index, as the usual standard error: *"a fund with a tracking error of 0.50% will be expected to perform within half a per cent of the index two-thirds of the time and within one per cent of the index 95% of the time"*. Note that this also assumes that the returns follow a normal law. Bishop (1990b) also defines term "passive fund management" (or simply *passive management*). *"First, the index fund can be passive in the sense that it is only subject to low levels of turnover"*. This means that the index fund is not expected to be changed very often — at least not before a customer of the index portfolio demands its money. Bishop (1990b) continues: *"The second interpretation of passive fund management is the technique

for allowing the market to take decisions for the fund manager"*, for example by following an index.

Having established these preliminary definitions, Bishop (1990b) addresses the reasons for actually choosing an indexed portfolio. Bishop first mentions the theoretical arguments. These are related with both the efficient markets hypothesis and the capital asset pricing model; two concepts that we presented in sections 4.4 and 4.5. Very roughly stated, the theory says that attempts to beat the the market will be ineffective because the price at which a stock can be bought or sold is "efficient" — there are no free lunches. So if the market is generally efficient, it is possible to beat the average consistently only if:

- the fund manager is lucky or
- the fund manager has access to inside information or
- the fund manager concentrates his efforts on a less efficient area of the market such as smaller companies.

Recall also the the capital asset pricing model and the capital market line with its linear risk-return relationship. So if an investor takes on additional risk, then he should be compensated for carrying that risk by receiving a higher return. Hence, if we consider an index fund as the form of equity investment that is exposed to the lowest risk (because all non-systematic risk inherent in single stocks is diversified away), then any non-index strategy carries a higher risk. Bishop (1990b) continues: *"Hence the non-index strategy should also produce a higher return for the investor, or else he is carrying risk for which he is not being compensated. [...] The overwhelming evidence is that the active investor is not receiving value for the additional risk being carried over and above that of an index fund. Indeed, research carried out [...] suggests that the number of managers outperforming the index each year could be explained by chance rather than ability."*

Next, Bishop considers performance arguments that she divides into two distinct groups — those that relate to the index itself and those that relate to the performance of index funds. The index itself yields higher returns than the most portfolios managed by actively selecting managers. One usually refers to transaction costs as the cause of this performance difference — of course, the theoretical benchmark is not exposed to real transaction costs. Index funds, on the other hand, come close to the return on the benchmarks, the index. And as Bishop points out, *"the fundamental point is that they yield this same steady rate of return consistently each and every accounting period"*. Thereafter, Bishop (1990b) evokes strategic arguments. Indexation comes with its own benchmark — the index. Finally, economic arguments are considered. As indexation requires less management overhead as active strategies, it gives (possibly higher) return at a lower cost.

Bishop (1990b) then turns towards the arguments brought forward against indexation. The first of these concerns market equilibrium: it would be detrimental to the market if a large proportion was to be locked into a highly predictable and static strategy so that the market would cease to be efficient. Bishop (1990b) turns this upside down: *"indexation has prospered because stock pickers have failed to identify the undervalued securities in an efficient market"*. The next counter-argument considers the allocation of capital. If indexation concentration continues, only companies represented in indices will be able to raise money. This overseas that venture capital always sought growth industries — so that there are already small company indices that even exclude bigger companies.

The next theme that Bishop (1990b) addresses in that of fund management issues where she starts to regard stock index futures. The aspects mentioned by Bishop concerning futures are the usual ones: that they provide a quick and relatively cheap (low margins compared to the premiums paid for options) way to "buy" the underlying index. Thus by buying future contracts with the cash not yet invested in stocks from the index, a fund manager can ensure that he is always completely invested in the index. What Bishop leaves out is that there are indices where there are not yet future contracts for, so that one cannot use index futures as alternatives for index funds. And, more important, a fund manager can always define its own index by deliberately choosing industries to be in- or excluded or similarly by concentrating on certain stock markets over the globe. Bishop (1990b) thereafter discusses the choice of an index. There are fundamental differences between capitalization weighted and arithmetic or geometric mean indices — we will discuss this when we introduce the *Deutsche Aktienindex* or *DAX*. Evidently, in order to diversify non-systematic risk, the index should be broadly based as the Standard and Poor's 500 Index.

5.3 On Index Fund Construction (Liesching and Manchanda 1990)

Liesching and Manchanda (1990) address the relevant aspects of how to set up a portfolio that is supposed to match an index. They first differentiate between visible and hidden costs. The former covers the management fees whereas the latter comprises establishment expenses and annual maintenance costs and they estimate that hidden establishment costs can take up about 0.5% of the fund value. But most important are establishment costs which can be over 1% of the portfolio value.

The first thing to do when setting up an index fund is to select a portfolio. This portfolio should match the index returns as closely as possible with minimum expense. Costs arise from turnover so that the choice of an index to be tracked is important. Liesching and Manchanda (1990) note that narrow indices (e.g. FTSE 100, Nikkei 225 or Dow Jones) are cheaper to track but that they have a coverage biased towards high capitalization stocks. Large institutional funds may want to get a broader market coverage. Narrow indices, on the other hand, have the advantage of higher liquidity. According to Liesching and Manchanda, *"most international index funds tend to be based upon such narrow liquid indices"*.

Liesching and Manchanda then discuss different approaches to constructing an index fund. For physical equities (as opposed to synthetic index funds based on index futures), there are two basic methods. The first is *replication* where all the stocks from the index are bought in the same proportions as in the index. This is the most expensive method which, for a broad index, will have a huge entry cost. The liquidity problem is important as purchasing the entire fund means dealing in small stocks with wide bid/offer spreads. As well, buying and selling can be delayed and have impacts on the share's price. According to Liesching and Manchanda, it can cost well over 2% to go from cash to a fully replicated index fund. Furthermore, these funds are difficult to maintain. Cash flows have to be divided between all the shares, e.g. between 683 stocks for the UK FTA All Share index. This results in

a high number of small trades: *"back office nightmare"*. However, the advantage of fully replicated funds is that the index will be closely tracked. For narrow indices, this could be an appropriate method.

The next method is *sampling* where the idea is to select fewer shares which still track the index closely. So turnover is reduced, stock with little turnover can be avoided and pre-existing portfolios can be used. According to Liesching and Manchanda (1990), there are two main issues to sampling. Stock specific effects have to be minimized by diversifying sufficiently and biases in the stock sample have to be avoided. Sampling can be further divided into *stratified sampling* and *optimised sampling*. With the stratified sampling approach, one divides companies into groups by size of market capitalization and sector. The idea is then to select shares from each group so that the resulting portfolio has the same percentage in each group as the benchmark index. These funds have a track record of almost matching index funds for the UK over the last three to four years as Liesching and Manchanda point out. For the dimension of stratified sampling funds, they state a number of 350 stocks out of the 683 for the FTA All Share Index. The 100 biggest stocks are thereby bought in exact proportions as in the index and the remaining ones are selected as to match the sector and size groups chosen. This selection is not guided — so the fund relies on its managers experience. These funds are thus similar to active funds as they require judgement in order to achieve the tracking results. As well, there is no way to predict future tracking error. So the trade-off between costs and tracking accuracy cannot be managed optimally. Concluding, one finds that this method is rather simple but requiring a lot of judgement.

Optimised sampling, as defined by Liesching and Manchanda (1990), uses factor analysis models based upon large amounts of historical information to set up an index fund. An index portfolio is then chosen as to guarantee the same exposure to these factors as for the index. The underlying model allows to calculate a predicted tracking error (which Liesching and Manchanda define as a predicted annualized standard error). This allows to adjust the accuracy of tracking as well as the costs of restructuring. The modeling also enables the fund manager to simulate different portfolios in order to find appropriate solutions; existing portfolio can also be accommodated. The factor models either use abstract statistical factors or fundamental factors as interest rates. These latter are more popular with fund managers and clients. Liesching and Manchanda refer to the mid-70s work of Rosenberg and of Ross as the underlying theory. The factors are often divided into common factors and sector factors where there are of course overlaps between them. The factors should be as independent as possible (orthogonal in mathematical terms). The main criticism is that these factor models assume that the way how factors affected performance in the past will be the same in the future. According to Liesching and Manchanda, well-designed factor models will use statistical techniques to identify these relationships. Models can be re-estimated and tested as well. As an example, Liesching and Manchanda state that the model from BARRA (a US-based financial consulting firm specialized in indexation) uses twelve common factors and over thirty sector factors for their UK equity model. Setting up an index fund by a factor model can be done by examining the factor deviation of an existing portfolio and then to select the best stocks to buy and sell using the computerized model. This procedure can be tailored to take account of client specific needs (e.g. a company may not be allowed to hold its own stocks or foreign investors might be restricted to certain amounts of stocks for different companies) or market circumstances. As a result, with 250 to 300 stocks a tracking error of under 0.5% can be obtained for the UK. The main benefit of optimised sampling is this good level of tracking accuracy as well as the flexibility obtained by using the factor models. This means that this method is the most cost efficient way of creating index funds. The disadvantage of this method is, according to Liesching and Manchanda, that the models are rather complex.

Optimised sampling can be used for multi-market indexation as well. As for single markets, it is not necessary to hold every stock in a multi-market indexation approach. Markets as a whole can have a very strong relationship (e.g. the Austrian and the German stock markets) so that one can obtain good tracking results without having to invest in every market. For multi-market models, Liesching and Manchanda (1990) say that one first estimates with single-market models as described above and then aggregates the results into a multi-market factor model. These models usually have fewer factors as there is less information available in smaller and less sophisticated models (e.g. the BARRA model for the FTA World Index only looks at four factors).

Another indexation approach is *synthetic index funds* where derivative instruments as stock index futures are used instead of physical equities. This permits fast and inexpensive trading as there are less transaction costs and no transaction taxes. Futures markets are usually very liquid so that transactions can be executed immediately; this makes it attractive for asset allocation purposes where one might have to disinvest a position rapidly. The major disadvantage of the synthetic index funds is that they do not track as closely as the other methods discussed above. There are several sources of tracking error. First, the underlying future might be for another market index. In the UK, the only available index future is on the FTSE 100 whereas the FTA All Share might be the benchmark. These two indices have a tracking error of approximately 2%. For the japanese market, the difference is even of 5% between the Nikkei 225 and the TOPIX as the former is a price-weighted index and the latter a capitalization-weighted one. Second, rolling of the futures contract (i.e. sell the near contract and buy the next contract) can lead to positive or negative tracking error. Third, liquidity might still be insufficient for large contracts in some markets so that this will move the market against the investor. Fourth, stock index future valuation depends on dividend forecasts — that might be wrong. Fifth, if the cash in the portfolio earns less than the rate used for the future valuation, then there will be underperformance of the index fund. On the other hand, a smart cash management might return more than the money market rate used for the fair-value calculation used for the future so that there might be overperformance as well. Summing up, synthetic index funds are popular when there is emphasis on speed and low transaction costs as in asset allocation.

Liesching and Manchanda (1990) discuss then trading methods as cross-trades or basket-trades to reduce costs further; we will not discuss this at this study focuses on the stock selection problem.

5.4 More on Index Fund Construction (Meade and Salkin 1989)

Meade and Salkin (1989) also address the selection procedures for the construction of index funds. They first define an index fund as *"one designed to reproduce the behaviour of a chosen market index"* and point out that its attraction is that the only risk associated with

investment in an index fund is market risk as specific company risk is diversified away. They also state that there is little substantial information in the literature about index funds. They then outline three basic approaches to index fund construction.

- Full replication — all shares in the index are held in the same proportion as in the index where there are high set-up, maintenance and divestment costs associated with this strategy; this is compensated for by good tracking of the index and little requirement for managerial judgement as we saw in the last section. Meade and Salkin also point out the obvious fact that a method other than full replication exposes the fund to the effects of various types of share activity; anomalous behaviour of the company's share price included in the index but not in the fund will cause a deterioration of the tracking record.
- Stratification — shares are selected to achieve the same sector representation and proportional capitalization as the index, but ignoring companies below a given level capital. This method has, according to Meade and Salkin (1989) similar, but scaled down, advantages and disadvantages to full replication. Liesching and Manchanda (1990) call this method "stratified sampling".
- Sampling — here shares of a small number of companies are selected, rather than sampled, to match the overall behaviour of the market. For Meade and Salkin, the advantages of this approach are lower set-up and maintenance costs. The crucial assumption is that historical relationships between share prices and index values persist. Liesching and Manchanda (1990) called this method "selection" rather than sampling.

Meade and Salkin address the measurement of tracking performance next. As they claim not to have found a definition of the tracking error, they provide one. Therefore, they first define for a moment t the value of the index fund as P_t and for the index itself as I_t. They then give a definition for the return over a period with length L as

$$\begin{aligned} p_{t,L} &= (P_t - P_{t-L})/P_{t-L} && \text{for the index fund and} \\ d_{t,L} &= (I_t - I_{t-L})/I_{t-L} && \text{for the index.} \end{aligned}$$

The tracking error over L time periods is then the root mean square deviation between these returns:

$$R_L = \left[\sum_{t=L}^{T} (d_{t,L} - p_{t,L})^2 \right]^{1/2} ,$$

where there are T observations of index and index-fund values. This tracking error definition is a generalization the usual return concept which is included here for a lag $L = 1$. The obtained value for the tracking error is dependent on the frequency of the observations and the value of L. As well, it is important to ensure that when one compares tracking error values that the bases are the same: a tracking error measuring returns over a week calculated with daily data will not necessarily be the same as that calculated using weekly data.

Having established these definitions, Meade and Salkin (1989) turn towards the construction of index funds. They recall: *"The overall objective is to construct an index fund which minimizes a specified tracking error using the shares of as few different companies as possible.* [...] *The underlying assumption of the selection approach is the same as that of short-term* *forecasting, i.e. that the underlying statistical relationships that are being modeled by the selection of the index fund will persist."* Their starting point are two desirable 'properties' that a fund might possess:

- stratification — the fund should have a proportion invested in each industrial sector similar to that in the index.
- capitalization weighting — the representation of a share in the index fund is proportional to the valuation of the company. This preserves the relative importance of companies.

Meade and Salkin (1990) then identify four construction methods:

Estimated coefficients — non-stratified This is an unconstrained minimization of the tracking error. Defining the share price of company i among the N companies available for selection at time t as X_{it}, the value of the fund at time t is

$$P_t = \sum_{i=1}^{N} \phi_i x_{it} ,$$

where $x_{it} = X_{it}/X_{i1}$ is the normalized share price. Using the definition for p_t given above, the objective is to choose $\phi_i \; \forall\, i = 1, \ldots, N$ such that

$$R_1^2 = \sum_{i=1}^{N} (d_{t,1} - p_{t,1})^2$$

is minimized; this is of course equivalent to minimizing the lag $L = 1$ tracking error. The constraints are

$$\sum_{i=1}^{N} \phi_i = 1 \quad \text{and} \quad \phi_i \geq 0 \;\; \forall\, i = 1, \ldots, N ,$$

the fund should be fully invested and does not issue liabilities. Expanding R_1^2 shows that the ϕ_is occur in the denominator and in the numerator. To simplify this optimisation, Meade and Salkin approximate the value of the index fund with the value of the index. This enables them to solve the problem as a quadratic programme. This assumption is not obvious! They replace an expression with another — but as the replaced one ought to be an approximated substitute for the replacing one, they simply take their desired result (that the approximation is in fact "close") as given! So a simple assumption permits Meade and Salkin to solve their problem numerically, but it is not entirely clear that this is an assumption that ought to be made.

Estimated coefficients — stratified For stratifying the fund, the proportion of the index represented by each stratum need to be known. Strata are typically chosen as industrial sectors. Assuming that there are S strata, we define θ_j as the proportion of stratum j in the index where

$$\sum_{j=1}^{S} \theta_j = 1 \quad \text{and} \quad \theta_j \geq 0 \;\; \forall\, j = 1, \ldots, S .$$

Each company is then associated to one stratum. The constraints

$$\sum_{i=1}^{N} \phi_i \delta_j(i) = \theta_j \;\; \forall\, j = 1, \ldots, S \quad \text{where} \quad \delta_j(i) = \begin{cases} 1 & \text{if firm } i \text{ belongs to stratum } j \\ 0 & \text{otherwise} \end{cases}$$

are then added to the problem just described which is again solved approximately as a quadratic programme.

Capitalization-weighted — non-stratified Here the proportion invested in a company is predetermined if the company is selected. Assuming that at time t there are W_{it} shares outstanding for company i, Meade and Salkin (1989) define

$$\pi_i = \begin{cases} 1 & \text{if company } i \text{ is a member of the fund} \\ 0 & \text{otherwise} \end{cases}$$

and get hence for the value of the index fund at time t

$$P_t = \frac{\sum_{i=1}^{N} \pi_i W_{it} X_{it}}{\sum_{i=1}^{N} \pi_i W_{it}} .$$

The objective is to choose the π_i to minimize R_1^2. This yields a zero-one integer programme with a fractional quadratic objective function that they try to solve, in the absence of a direct solution method, with an iterated heuristic method.

Capitalization-weighted — stratified Such a fund cannot, owing to the lack of flexibility, be exactly stratified. The constraints that enforce stratification are modified as follows:

$$\frac{\sum_{i=1}^{N} \pi_i W_{i1} X_{i1} \delta_j(i)}{\sum_{i=1}^{N} \pi_i W_{i1} X_{i1}} + H_j^+ - H_j^- = \theta_j \;\; \forall\, j = 1, \ldots, S$$

where

$$H_j^+ \cdot H_j^- = 0 .$$

H_j^+ and H_j^- represent the positive and negative violation of the stratification constraint. The objective to be minimized is

$$R_1^2 + \tau \sum_{j=1}^{S} (H_j^+ + H_j^-)$$

where τ is positive scaling factor. This problem is then, as the preceding one, solved approximately by an iterative technique.

Before testing these four approaches on financial markets data from Japan, Meade and Salkin (1989) discuss the relation of the index-fund selection to the portfolio selection of the usual Markowitz-type. This latter can be written as

$$\text{minimize } V(p_{t,L}) \quad \text{subject to} \quad E(p_{t,L}) = E(d_{t,L}) ;$$

this class of problems is usually solved by quadratic programming techniques. On the other hand, the index-fund selection requires that the risk and return of the fund matches that of the index as closely as possible over any investment horizon. The minimization of R_1^2 implies that

$$V(p_{t,L}) = V(d_{t,L}) \quad \text{and} \quad E(p_{t,L}) = E(d_{t,L})$$

are satisfied as far as possible, but the more stringent requirement implied by this problem is that

$$\operatorname{cov}(p_{t,1}, p_{t+k,1}) = \operatorname{cov}(d_{t,1}, d_{t+k,1}) \quad \forall k$$

should be satisfied. For index-fund selection, there is an additional requirement in the time domain, namely that the returns on the index fund possess a similar auto-covariance structure.

Meade and Salkin (1989) thereafter present their results of tests of these four approaches for the Japanese stock market. They had weekly data from January 1985 until September 1987 for the hundred biggest companies and they tried to track the TOPIX index that comprises over 1000 shares. (Note that this of course implies that they could not achieve a tracking error of zero as there are 900 shares missing.) Their analysis consisted of the construction of an index fund using the four methods discussed above. In each case the tracking error was calculated for returns over 1,2,4 and 8 weeks over the 2 years used for the construction of the fund (the in-sample period). The same set of tracking errors are calculated for the first 3, 6 and 9 month periods of data in 1987 (the out-of-sample period).

Summarizing the in-sample performance, Meade and Salkin (1989) find that the the ranking in terms of R_k for $k = 1, 2, 4, 8$ corresponds in most cases to the method number. In all cases, method 1 (no constraints) gives the best performance; method 4 (with two sets of constraints) gives the poorest. For out-of-sample performance, several patterns emerged from examination of these results. Unsurprisingly, the values of R_k were all greater than the corresponding in-sample values. In all cases, the values of R_k for each method improved as the period over which they are calculated increases. The overall best performing method was the first, unconstrained estimation. This method achieves the lowest values of R_k for $k = 1, 2, 4, 8$ for nine out of the 12 cases examined. In the three other cases, it achieves the second lowest values. These results indicate that, on the data used, no benefits in terms of better tracking performance were gained by constraining the index fund. The constraint of capitalization weighting carried less penalty than that of stratification and the imposition of both constraints led to the worst overall performance. Unfortunately, Meade and Salkin fail to give any significance test result for the comparison of these four methods.

Meade and Salkin then consider the correlation of the tracking errors. They state that, if the tracking errors $d_{t,1} - p_{t,1}$ are uncorrelated with each other, one would expect that

$$E(R_8^2) = 2 \cdot E(R_4^2) = 4 \cdot E(R_2^2) = 8 \cdot E(R_1^2)$$

holds. A more sensitive means of determining whether the tracking errors are uncorrelated is to examine the autocorrelation function (a.c.f.) of the errors. Meade and Salkin then give a.c.f. values for the tracking errors over all the data and all four methods of fund construction. The first method shows marginally significant correlations at the 5% level at lags of 1,4,5,6 and 9 weeks. The other three methods exhibit one significant correlation each. Meade and Salkin (1989) conclude thus that the relation

$$R_K^2 = K \cdot R_1^K$$

to estimate risk over the appropriate investment horizons could lead to incorrect estimates as the underlying assumption of non-autocorrelated differences in returns is not generally valid. Concluding, Meade and Salkin (1989) summarize that the imposition of the constraint led to a deterioration in performance; straight optimisation emerged as the best performing method.

5.5 Maintaining Index Funds (Meade and Salkin 1990)

In their sequel, Meade and Salkin (1990) start by giving a new definition for the tracking error as a root-mean-square error measure of the difference in returns between the portfolio and the index:

$$TE = \sqrt{\sum_{t=1}^{T} \frac{(F_t - I_t)^2}{T}}\,, \tag{5.1}$$

where, at time t, the return on the chosen market index is I_t, the return on the index fund is F_t, and T is some time horizon.

The problem that Meade and Salkin address is to develop policies that minimize the running costs of an equity index fund. They assume that complete or partial replication of the index is not a viable strategy, and the method of construction of the equity fund requires the choice of a subset of shares from those available. Refering to their preceding paper, they chose direct estimation of the coefficients as the selection method. Taking into account the non-constant variance of the noise terms for financial asset prices, they model the returns on these assets using an ARCH approach following the papers of Engle (1982), Bollerslev (1986) and Taylor (1989). Taylor used an ARCH model with an autoregressive trend term for the simulation of futures prices.

The form of the model is:

$$\begin{array}{rcrcll} X_{i,t} & = & \mu_{i,t} & + & \nu_{i,t} & \forall\, i = 1, \ldots, N \\ I_t & = & \mu_{N+1,t} & + & \nu_{N+1,t} & , \end{array}$$

where $X_{i,t}$ is the return on share i at time t and I_t is the return on the index at time t. The noise terms are assumed to satisfy the following requirements:

$$E(\nu_{i,t}) = 0 \ \text{ and } \ \operatorname{cov}(\nu_{i,t}, \nu_{j,t}) = \sigma_{i,j,t} \ \forall\, i = 1, \ldots, N+1;\ j = 1, \ldots, N+1$$

and

$$\operatorname{cov}(\nu_{i,t}, \nu_{j,t+k}) = 0 \ \ \forall\, k \neq 0 \,.$$

The mean return on share i, $\mu_{i,t}$, and the covariance between the returns on share i and share j, $\sigma_{i,j,t}$, for $i = 1, \ldots, N+1$ and $j = 1, \ldots, N+1$ are assumed to change slowly over time. That is

$$\begin{array}{rcrcl} \mu_{i,t} & = & \mu_{i,t-1} & + & \varepsilon_{i,t} \\ \sigma_{i,j,t} & = & \sigma_{i,j,t-1} & + & \eta_{i,j,t}\,, \end{array}$$

where

$$\begin{array}{ll} \operatorname{cov}(\varepsilon_{i,t}, \varepsilon_{i+l,t+k}) & = 0\,, \qquad \text{for } l \text{ or } k \neq 0 \ \text{ and} \\ \operatorname{cov}(\eta_{i,j,t}, \eta_{i+l,j+k,t}) & = 0\,, \qquad \text{for } l \text{ or } k \neq 0\,. \end{array}$$

Taylor suggested to overcome the effect of fluctuating variance of $v_{i,t}$ in

$$X_{i,t} = \mu_{i,t} + v_{i,t}$$

by rescaling the time series in dividing by

$$a_{i,t} = E(|X_{i,t}|) ,$$

$a_{i,t}$ is roughly proportional to $\sqrt{\mathrm{var}(v_{i,t})}$ (the mean $\mu_{i,t}$ is neglected as it is close to zero). After division, one obtains the new time series

$$Y_{i,t} = \mu'_{i,t} + v'_{i,t}$$

where $\mathrm{var}(v'_{i,t})$ is now constant over time. Thus, if $a_{i,t}$ is estimated by

$$\hat{a}_{i,t} = \gamma \sum_{k=0}^{\infty} (1-\gamma)^k |X_{i,t-i-k}| ,$$

then

$$\hat{\mu}'_{i,t} = \alpha \sum_{k=0}^{\infty} (1-\alpha)^k \frac{X_{i,t-1-k}}{\hat{a}_{i,t-1-k}}$$

and

$$\hat{\mathrm{cov}}(v'_{i,t}, v'_{j,t}) = \alpha \sum_{k=0}^{\infty} (1-\alpha)^k (Y_{i,t-1-k} - \mu'_{i,t-1-k}) \cdot (Y_{j,t-1-k} - \mu'_{j,t-1-k}) .$$

Meade and Salkin (1990) focus thereafter on the maintenance cost aspect of equity index funds. They define $W(t_1, t_2)$ as the expected cost of maintaining an index fund from time t_1 to t_2 where they suggest that this is proportional to the square of the difference between the return on the index and the return on the index fund. If the fund is to be run for L periods without readjustment, the fund manager would want to minimize the expected cost of operation from $t = 0$ until $t = L$:

$$W(0, L) = c_1 E \left(\sum_{t=1}^{L} (F_t - I_t)^2 | \Theta_0 \right)$$

where Θ_0 represents the information available at time t; c_1 is a positive scaling factor. As in the preceding section, we have

$$F_t = \sum_{i=1}^{N} \phi_{i,0} X_{i,t}$$

where $\phi_{i,0}$ is the proportion of the fund invested in share i at the beginning of the period. This gives the constraints

$$\phi_{i,0} \geq 0 \;\; \forall\, i = 1, \ldots, N \quad \text{and} \quad \sum_{i=1}^{N} \phi_{i,0} = 1 .$$

Substituting, expanding and rearranging yields

$$\begin{aligned} W(0, L) \;\; = \;\; & c_1 \left[\sum_{i=1}^{N} \sum_{j=1}^{N} \phi_{i,0} \phi_{j,0} \sum_{t=1}^{L} E(X_{i,t} X_{j,t} | \Theta_0) \right] \\ & - 2c_1 \left[\sum_{i=1}^{N} \phi_{i,0} \sum_{t=1}^{L} E(X_{i,t} I_t | \Theta_0) + \sum_{t=1}^{L} E(I_t^2 | \Theta_0) \right] . \end{aligned}$$

Following the model for the return variances, we can simplify with

$$\begin{aligned} E(X_{i,t} X_{j,t} | \Theta_0) &= \mu_{i,0} \mu_{j,0} &+ \;\; \sigma_{i,j,0} \\ E(X_{i,t} I_t | \Theta_0) &= \mu_{i,0} \mu_{N+1,0} &+ \;\; \sigma_{i,N+1,0} \\ E(I_t^2 | \Theta_0) &= \mu_{N+1,0}^2 &+ \;\; \sigma_{N+1,N+1,0} . \end{aligned}$$

The problem of choosing $\phi_{i,0}$ $\forall\, i = 1, \ldots, N$ to minimize $W(0, L)$ subject to the constraints is now a straightforward quadratic programming problem that Meade and Salkin (1990) solve with a numerical solutions package.

Meade and Salkin (1990) test this approach on data for the 100 biggest companies from the London Stock Exchange for the period January 1984 to April 1989. The index to be tracked is the Financial Times/Actuaries 500 (or FTA 500 for short). They construct the first index fund after a three months interval and thereafter readjust every three months. As their interest is index fund maintenance, they define a cost function

$$C(T) = \sum_{t=1}^{T} |\phi_{i,T} - \phi_{i,T-L}|$$

that is directly proportional to the number of shares bought and sold where L is the sub-period length for construction/readjustment (e.g. 13 weeks). Meade and Salkin (1990) comment their results: *"The tracking error performance is erratic. The root mean-square error moves between 0.23% per week 0.92% per week (over a full 13 weeks) without showing any strong tendency to approach equilibrium. The transaction costs indicated by $C(T)$ vary between 0.62 and 1.52, showing that at each readjustment, between 31% and 76% of the fund is redistributed. Both aspects, uneven tracking performance and dramatic redristribution of the fund's investment, are unsatisfactory. Two issues are worth raising at this stage. First, at each solution of the quadratic programme, there appears to be an almost flat region around the optimal solution. This implies that there are a large number of possible selections for the index fund which are almost as good as the optimal solution. Some of these solutions may be more stable than the optimal solution, in the sense that their tracking performance may be more consistent. Secondly, since it is capitalization weighted, the index is more sensitive to the movements of share prices for the larger companies."*

Following this second observation, Meade and Salkin (1990) then force the 25 largest companies into the index fund in order to achieve greater tracking stability. This reduced the mean tracking error from 0.59% per week to 0.49% per week, and the total transaction cost is reduced from 12.8 to 8.7. Meade and Salkin then test values of $L = 8, 10, 26$ and 52 weeks for the sub-period length. Generally, as the readjustment period increases, total transaction cost tend to decrease, and, correspondingly, the long-run tracking error tends to deteriorate.

Meade and Salkin then consider more flexible approaches to readjustment; the first one is to readjust if

$$\sqrt{\frac{1}{N}\sum_{t=T}^{T+N}(F_t - I_t)^2} > \delta_1 \ ,$$

i.e. when the tracking error exceeds a certain limit that is given as a constant δ_1; here T is the time when the fund was last adjusted and $N_{\min} \leq N \leq N_{\max}$ provides that the fund is not adjusted too frequently which would surely result in a deterioration of the transaction costs. A second criterion for readjustment that Meade and Salkin (1990) propose is

$$\frac{1}{N}\sum_{t=T}^{T+N}(F_t - I_t) < -\delta_2 \ ,$$

i.e. when the return from the index fund underperforms the return from the index more than the given level δ_2. One now has as well a parameter setting problem what Meade and Salkin see as well: "*The setting of δ_1, δ_2, $N_{\min}$ and $N_{\max}$ involves trading off transaction cost, tracking error and penalty from underperformance. The objective is to let a fund continue if it is performing well and to detect a poorly performing fund as soon as possible. This objective is not easily achieved since a poorly performing fund may improve; similarly, the performance of a fund that has tracked well for a long period may suddenly deteriorate*". The results that Meade and Salkin (1990) obtain for the variation of some values let the tracking error rather unchanged compared with the 'regular adjustment after ten weeks' policy; transaction costs seemed to ameliorate but the authors give no significance values. But Meade and Salkin (1990) remark one interesting outcome: $\bar{F} - \bar{I}$ is always positive, the index fund had a constantly higher return than the index. The authors leave the question whether this is due to the share return model or the data set used unanswered.

Lastly, they discuss the possibility of adding further constraints to the index fund construction problem. The first constraint that they evoke is

$$\sum_{i=1}^{N}\phi_{i,t}\beta_i = 1.0$$

where the β_i is Sharpe's beta-coefficient that we introduced in the previous chapter. This further forces the index fund to look like the market index that is taken as an approximation for the market portfolio. The second constraint is

$$\sum_{i=1}^{N}\phi_{i,t}y_{i,t} = y_{N+1,t} + \Delta$$

where $y_{i,t}$ is the dividend yield on share i at time t, $y_{N+1,t}$ is the corresponding value for the index and Δ is the premium required by the fund manager. This should control dividend income from the fund; the application of Δ is called "tilting the fund". Meade and Salkin (1990) do not give any results for these two strategies.

Concluding, Meade and Salkin (1990) point out that triggering the readjustment by performance has lowered transaction costs, but that further knowledge is needed for the trade-off between transaction costs and tracking error. For further research, they point out the direction of multivariate modeling of returns as well as the one of the stability of the covariance structure of share returns.

5.6 Summary

This chapter provided an overview for the subject of indexation or passive management as it is equally called. We have seen

- with the introduction of Bishop (1990) what indexation is about, which preliminary definitions had to be made, which arguments account in favour of indexation and what can be said about the argument against indexation;
- with the chapter from Liesching and Manchanda (1990) on index fund construction, we were introduced rather informally to three basic indexation methods: replication, stratified sampling and optimised sampling; Liesching and Manchanda (1990) give optimised sampling, which is based on multi-factor models, the edge over the other methods;
- with the first paper from Meade and Salkin (1989), we further discussed index fund construction methods in a more profound way by explicitly specifying the problems to be solved; Meade and Salkin see direct estimation (i.e. optimisation) of the coefficients as the clear winner; unfortunately, we have no comparison of direct estimation and multi-factor model based optimisation;
- with the following paper by Meade and Salkin (1990), we discussed further aspects of index fund construction; this time under the aspect of fund maintenance; as well, we considered in depth their model of share returns.

It might be interesting to compare the optimised sampling approach that is based on a multi-factor model with the Meade and Salkin approach of coefficient estimation. Unfortunately, there is no paper available that undertakes this comparison. We leave this chapter hence with two different recommendations for two different 'best' methods. On the hand, among the papers that we considered, none recommended the stratified sampling or the full replication approach.

It is worth pointing out at this stage that all of methods used by Meade and Salkin (1989) were heuristic. The indexation problem does not know a direct solution method. We can divide the problem into two stages: selection and weighting. One first has to select a subset of shares that the index fund will be constructed of; thereafter weighting tries in the second stage to find the combination of these shares that comes as close as possible to the index for a certain criterion (say, minimal standard error between the two). Numerical analysis could provide us with exact methods for either of the problems (integer programming; quadratic programming), but the combination of the two stages breeds a new problem. The absence of a solution method for this new problem makes it a perfect setting a genetic algorithm: a complex search space that does not provide a direct solution method.

Chapter 6

On the DAX

Indexation makes simple promises,
and it keeps those promises.
E. BISHOP (1990b)

6.1 Introduction

The *DAX* or *Deutsche Aktienindex* is the main index for the German stock market in Frankfurt am Main. We will in this chapter discuss the concepts of other major stock market indices and of course that of the DAX. We will see how capital changes and dividend payments are reflected in DAX value. We will thereafter regard how we can simplify its calculation. Last, we will see of which shares the DAX is actually composed and which part every of these takes in total DAX volume.

6.2 The Concept of the DAX

The DAX is a real-time, capitalization-weighted index of 30 major German stocks traded at Frankfurt am Main, Germany's principal stock market. Before we regard this benchmark, we will shortly consider the basic concepts of index calculation and the major indices for the biggest stock markets. The history of stock market indices for the US goes back to 1884. Tomlinson (1990) points out that these first indices were designed to reflect movements in the market over short periods of time; they were not intended for long time use. The first index was a simple arithmetic average. When a stock went ex-rights, continuity was maintained by adjusting to a cum-rights price on a stock-by-stock basis. This method was later replaced by introducing a "divisor" that divided the sum of the stock prices for the actual index value. This is the method used for the Dow Jones Index series, which includes the well known Dow Jones Industrial Average. The drawback of these price-weighted indices is that changes in a high-priced stock have a greater influence than a comparable proportionate price change in a low-priced stock. Furthermore, the market capitalization of individual stocks is not taken into account which *"seems to make little sense for a long-term market barometer"* (Tomlinson 1990). The response came 1923 with the Standard & Poor's indices which sum the market

capitalizations of its stocks (e.g. the 500 that form the "S&P 500") and divide then by a divisor that is continuously adjusted for capital changes in the stocks. In Japan, the Nikkei Average that comprises 225 stocks has been calculated since 1949 using the Dow formula that we described above. Since 1969, the Tokyo stock exchange calculates the index that is now called TOPIX. Similar to the S&P 500, the TOPIX is a market capitalization-weighted average of about 1200 stocks and the definitive benchmark for Japanese equity investors (Tomlinson 1990). In the UK, the Financial Times (FT) 30 Share Index, that dates back to 1935 and that is calculated as a geometric average, used to be the major index. This method has two problems. First, geometric indices always underperform arithmetic indices. Suppose for an example that an index is constituted by two shares every of which has a price of 100. As $1/2(100+100) = (100 \cdot 100)^{1/2}$, both the arithmetic and the geometric formula give hence 100 as the index value. Suppose as well that sudden interest in the first share gives raise to a ten per cent increase to 110 for its share price. Arithmetically averaging, we obtain $1/2(110+100) = 105$ whereas the geometric average underperforms: $(110 * 100)^{1/2} = 104.88$. The second and actually more more serious problem is that the whole index goes to zero that if a stock becomes insolvent, So in 1962 the market capitalization-weighted FT-Actuaries All Share Index (FT-A All Share for short) was introduced. It initially contained 750 stocks, Tomlinson (1990) gives 687 as its number of constituents. This is the definitive benchmark for the UK equity market although the smaller FTSE 100 which is calculated every minute is used for the stock index financial futures.

The DAX was introduced on July 1, 1988. It is a capitalization-weighted index comprising 30 shares that is calculated on a continuous basis, i.e. once a minute, during the official trading hours at the Frankfurt Stock Exchange. The shares are selected according to turnover, represented capital and an early availability of opening prices for each day. The DAX is called a *performance index* (Frankfurter Wertpapierbörse AG 1990, Richter 1990) or as well *total return index* (Richter 1990) because it is assumed that every cash flow generated by one of the stocks as for example from dividends or subscription rights is completely reinvested in more stocks by the same company. That implies that the relative weights of the companies in that index are not constant over time. We will consider the adjustment for these reinvestments in the index shortly.

The DAX at an instant t is defined as (we slightly changed the index notation from Frankfurter Wertpapierbörse AG 1990, Richter 1990, page 4, Loistl 1990, page 75)

$$DAX_t = K_{t1} \cdot \frac{\sum_{i=1}^{N} p_{i,t} \cdot q_{i,t1} \cdot c_{i,t}}{\sum_{i=1}^{N} p_{i,t0} \cdot q_{i,t0}} \cdot 1000 \tag{6.1}$$

where
$t0$ = time index for 30/12/1987
$t1$ = time index for 03/09/1990
$p_{i,t}$ = stock price of company i at day t
$q_{i,t1}$ = capital of company i at 03/09/1990
$p_{i,t0}$ = stock price of company i at 30/12/1987
$q_{i,t0}$ = capital of company i at 30/12/1987
$c_{i,t}$ = adjustment factor for company i with default value 1
K_{t1} = (constant) linking factor as of 03/09/1990.

The expression in the numerator is the sum over all the products of price, represented capital and the correction factor (that we will consider shortly in more detail) at each minute between 10.30h and 13.30h. Each of the $p_{i,t}\, q_{i,t1}$ products represents the market value of company i at date t, adjusted by the factor $c_{i,t}$ after a possible cash flow from that company. Note that multiplying by $q_{i,t1}$ cannot be replaced by multiplying by, say the actual number of shares outstanding as the German shares can have different nominal values; the by far most common is 50 DM per share, but there are shares with nominal values of 100 DM or even 500 DM. Using the capital in DM directly does avoid this difficulty. The denominator gives the sum of the products of closing prices and capital at the 30/12/1987, the last trading day in 1987; this is hence the total market value at the end of 1987. The DAX, that is actually calculated since July 1, 1988, was normalized in order to get a value of 1000 for December 30, 1987. With this base value of 1000, which facilitates calculation of margins for Index trading, the DAX is in line with other indices created in the last years: the FTSE 100 in London, the CAC 40 in Paris, the ISEQ in Dublin and the Toronto 300 Composite Index which was already introduced in 1977. Note also that the DAX is defined as the continuation of the *Börsen-Zeitung Index*, which itself is the continuation of the *Hardy Index*. It can therefore be dated back until the starting date for that latter index, which was September 28, 1959. Richter (1990) discusses in length the representativeness of the DAX for the whole German stock market, its number of stocks and their selection criteria.

The adjustment in case of capital changes or dividend payments is reflected in the factors $c_{i,t}$ using the expression

$$c_{i,t} = \frac{p_{i,t-1}}{p_{i,t-1} - v_{i,t}} \tag{6.2}$$

where $p_{i,t-1}$ is the closing price of the share in question on the day prior to the event and $v_{i,t}$ denotes the values of the event. Loistl (1991) derives this expression and gives as well several examples for the calculation of the $c_{i,t}$ and the $v_{i,t}$. If for example the possession of o 'old' shares gives you the right to receive n 'new' ones where $p_{i,t-1}$ is as above the closing price prior to the emission and $p_{i,n}$ the emission price for the new shares, then

$$v_{i,t} = \frac{n(p_{i,t-1} - p_{i,n})}{o+n} \,.$$

Substituting this into equation (6.2) yields

$$v_{i,t} = \frac{(o+n)p_{t-1}}{op_{i,t-1} + np_{i,n}} \,,$$

this is valid for all cases where $p_{i,t-1} \geq p_{i,n}$. For dividend payments, $v_{i,t}$ simply equals the payments (and one largely ignores income and capital taxes). Finally, if you receive n shares for free when you are holding o shares, the event value $v_{i,t}$ is

$$v_{i,t} = p_{i,t-1} - \frac{op_{i,t-1}}{o+n}$$

where $p_{i,t-1}$ is once again the closing price on the day before the free shares issue.

Supposing that at least one company earns enough money to pay its yearly dividend, we recognize that its factor $c_{i,t}$ will be changed once a year. Supposing additionally that this company pays a rather high dividend as for example the chemical companies BASF, Bayer

and Hoechst (which, with price of between 200 and 300 DM paid between 12 and 13 DM in the last years), we see that such a company would increase its index weight over the years and therefore distort the representativeness of the DAX. Therefore, usually once a year, all these adjustment factors are reset to 1, their normal values, while their respective values are taken into account by the linking factor K_t. This reset is called *chain linking* as well. It was executed for the first time about 15 month after the start of the DAX calculations on September 15, 1989. The second chain linking was undertaken on September 3, 1990. This day marks also the beginning of our data base. So the returns from a share are invested in the same share when they are received, but at a reset it is assumed that all reinvestments since the last reset are equally invested in an index portfolio. For the chain linking, the DAX value is established on the day prior to the chain linking according to

$$DAX_{t-1} = K_{t_{\text{old}}} \frac{\sum_{i=1}^{N} p_{i,t-1}\, q_{i,t_{\text{old}}}\, c_{i,t-1}}{\sum_{i=1}^{N} p_{i,t0}\, q_{i,t0}} \cdot 1000$$

where t_{old} is the previous linking date and $K_{t_{\text{old}}}$ the linking factor used since the last chain linking. Starting the next day, the DAX is calculated by using the actual share capital values $q_{i,t}$ and the prices $p_{i,t}$ as given but with all the $c_{i,t}$ reset to one. To avoid a jump in the DAX value for the opening value, the DAX_{interim} is calculated as

$$DAX_{\text{interim}} = \frac{\sum_{i=1}^{N} p_{i,t-1} q_{i,t}}{\sum_{i=1}^{N} p_{i,t0} q_{i,t0}} \cdot 1000 \,.$$

So one uses the share prices from the day prior to the chain linking date and the share capital values from the chain linking day; the adjustment factors $c_{i,t}$ do not appear as they are reset to 1. The new linking factor is then obtained via

$$K_t = \frac{DAX_{t-1}}{DAX_{\text{interim}}} \,.$$

Judging on the influence of capital changes and dividend payments, Richter (1990) writes that *"as a general rule, the dividend payments over-compensate the capital changes — usually capital increases — with the result that over time $K(t)$ rises steadily"*. This linking factor was originally set to the value 0.9646302 in order to establish 1000 as the basis of the DAX on December 30, 1987. Using the linking factor and the adjustment factors, it is also possible to change the underlying set of stocks without giving rise to a change is the DAX value. The last major change was executed on September 3, 1990 where Nixdorf AG and Feldmühle Nobel AG, which both were victims of takeovers by other DAX-quoted companies and therefore no longer representative in their own rights, were replaced by Metallgesellschaft AG and Preussag AG.

6.3 Calculating the DAX

We did specify with (6.1) a formula that allows to calculate the index value. In use at the Frankfurt stock market is another formulae that is a little easier to handle. We will derive this formula now (cf. Frankfurter Wertpapierbörse AG 1990). Starting with (6.1), we have

$$DAX_t = K_{t1} \cdot \frac{\sum p_{i,t} \cdot q_{i,t1} \cdot c_{i,t}}{\sum p_{i,t0} \cdot q_{i,t0}} \cdot 1000$$

and we will multiply the numerator by $\sum q_{i,t0}$ and $1/\sum q_{i,t0}$ and the denominator by $\sum q_{i,t1}$ and $1/\sum q_{i,t1}$, leaving the value of both parts of the fraction unchanged. Hence after a repositioning

$$DAX_t = \frac{\sum p_{i,t}\Big((q_{i,t1}/\sum q_{i,t1})\,100 \cdot K_{t1}(\sum q_{i,t1}/\sum q_{i,t0})\,c_{i,t}\Big)}{\sum p_{i,t0}\,(q_{i,t0}/\sum q_{i,t0}) \cdot 100} \cdot 1000\,.$$

We define

$$\begin{aligned} F_{i,t} &= (q_{i,t1}/\sum q_{i,t1}) \cdot 100 \cdot K_{t1}(\sum q_{i,t1}/\sum q_{i,t0})\,c_{i,t} \\ A &= \sum p_{i,t0}\,\Big(q_{i,t0}/\sum q_{i,t0}\Big) \cdot 100 \end{aligned}$$

and obtain

$$DAX_t = \frac{\sum p_{i,t} \cdot F_{i,t}}{A} \cdot 1000 \tag{6.3}$$

where $A = 29356.73$ is a constant and $F_{i,t}$ a share specific factor that takes into account the variation in the adjustment factors and the linking factors. The $F_{i,t}$ are published daily by the Börsen-Zeitung, the principal German newspaper for the stock exchange. So the professionals in the market only have to get these two values out of the newspaper or any other data source and plug them into their computer programmes without having to bother about chain linkings or even dividend payments. The prices $p_{i,t}$ as well as the factors $F_{i,t}$ together with the DAX values constitute thus our data base for the application part of this work. We are disposing of one year of daily prices $p_{i,t}$ and factors $F_{i,t}$, i.e. for the 247 trading days between September 3, 1990 and August 30, 1991. Figure 6.1 graphs the DAX during this one year period.

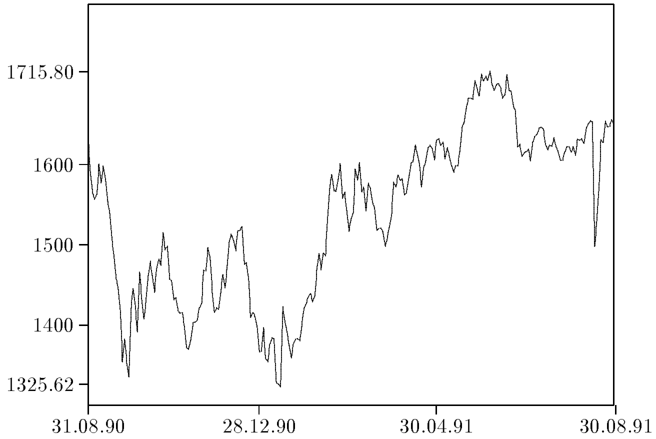
We did of course try to validate our data base. With equation (6.3) and the data base, we were able to recalculate the DAX for the period in question. Table 6.1 shows all days where the recalculated DAX lay outside a 0.05% band around the actual DAX value as given in the data base.

The slight inconsistency is probably due to the manual calculation of the $F_{i,t}$ by Deutsche Bank Research.

6.4 The Stocks in the DAX

The criteria for the selection of a company for the DAX were the turnover in a stock, the capital of that stock and the availability of early opening prices each day. These points are clearly complementary — stocks with high turnover will be traded all the time and thus with early opening prices; moreover, these stocks tend to be the important ones for a market in the sense of high capitalization. The constraint was of course to match as closely as possible the actual industrial structure in Germany. The idea was, as Richter (1990) writes, to include *"only as many index components as necessary but as few as possible, the index*

Figure 6.1: The DAX between September 3, 1990 and August 30, 1991



Source: The data was furnished by the Deutsche Bank Research GmbH.

was to be equated to a portfolio representing the market in its entirety and was therefore to form a suitable basis for derivative financial instruments, in concrete terms DAX futures,

Table 6.1: Verifying the DAX data base

Actual DAX value	Recalculated DAX	Error	Date
1422.35	1421.40	0.06677	04-Oct-90
1467.47	1464.67	0.19137	19-Nov-90
1587.13	1588.90	-0.11143	19-Feb-91
1605.79	1606.60	-0.05031	30-Apr-91
1630.04	1630.87	-0.05077	02-May-91
1631.84	1632.68	-0.05128	03-May-91
1607.28	1606.30	0.06121	08-May-91
1620.63	1619.65	0.06057	10-May-91
1610.90	1609.93	0.06015	13-May-91
1598.50	1597.54	0.05983	14-May-91
1590.35	1589.41	0.05912	15-May-91
1598.08	1597.13	0.05976	16-May-91
1598.87	1597.92	0.05920	17-May-91
1617.40	1616.43	0.06005	21-May-91
1647.69	1646.69	0.06062	22-May-91
1652.71	1651.28	0.08671	23-May-91
1682.90	1681.83	0.06384	27-May-91
1704.11	1697.30	0.40124	31-May-91
1700.84	1702.97	-0.12497	12-Jun-91
1655.50	1654.65	0.05135	29-Aug-91

Source: Own calculations
Note: Error is calculated as $100 \cdot (I_t - D_t)/D_t$ where I_t is the actual DAX value as in the data base and in column 1, D_t is the DAX value obtained with equation (6.3) as in column 2; we state all observations where Error exceeds 0.05.

Table 6.2: Stocks in the DAX

Company Name	short symbol	Capital per 03/09/90 in Mio. DM	Closing at 03/09/90 in DM	Capital per 30/12/87 in Mio. DM	Closing at 30/12/87 in DM	DAX per cent 03/09/90
Allianz	ALV	825.0	2570.00	750.0	1144.00	12.47
BASF	BAS	2849.8	235.50	2774.2	255.50	3.95
Bayer	BAY	3200.2	247.00	3121.3	263.80	4.65
Bay.Hyp.u.W'bank	BHW	827.3	359.50	750.6	333.00	1.75
Bay.Motorenwerke	BMW	790.6	484.00	750.0	447.00	2.25
Bay.Vereinsbank	BVM	617.3	382.00	493.0	316.00	1.39
Commerzbank	CBK	1272.7	263.50	1040.0	214.00	1.97
Continental	CON	445.5	300.00	402.4	205.20	0.79
Daimler-Benz	DAI	2327.4	699.00	2115.7	575.00	9.57
Degussa	DGS	365.0	348.00	365.0	316.00	0.75
Dt.Babcock	DBC	350.0	151.00	350.0	153.00	0.31
Deutsche Bank	DBK	2201.2	692.00	1772.6	388.00	8.96
Dresdner Bank	DRB	1819.7	413.00	1311.6	232.00	4.42
Henkel	HEN3	302.5	595.50	175.0	486.00	1.06
Hoechst	HFA	2920.0	237.50	2797.4	250.00	4.08
Karstadt	KAR	360.0	689.00	360.0	427.00	1.46
Kaufhof	KFH	478.6	590.00	389.2	417.20	1.66
Linde	LIN	291.6	880.00	238.0	522.00	1.51
Lufthansa	LHA	1520.0	130.00	1200.0	135.50	1.16
MAN	MAN	771.0	359.00	674.5	139.00	1.63
Mannesmann	MMW	1449.2	274.50	1273.4	101.10	2.34
Metallgesellsch.	MET	385.1	569.90	280.0	211.00	1.29
Preussag	PRS	709.4	346.00	401.6	114.00	1.44
RWE	RWE	2215.3	418.00	2213.3	209.50	5.45
Schering	SCH	341.1	652.00	283.0	353.00	1.31
Siemens	SIE	2534.2	626.00	2379.2	359.00	9.33
Thyssen	THY	1665.0	233.50	1565.0	104.50	2.15
VEBA	VEB	2226.7	354.00	1981.8	260.10	4.64
VIAG	VIA	789.0	354.00	580.0	179.10	1.64
Volkswagen	VOW	1650.0	475.00	1500.0	224.50	4.61
Sum		38401.0		34287.8		99.98

Source: Frankfurter Wertpapierbörse AG; own calculations.
Note: Henkel shares come in two qualities: ordinary shares with voting rights, all of which are held by the Henkel family, and preference shares without voting rights. These latter are the only ones traded. The shares for all other in the DAX are proper shares with all the usual rights. The difference to 100 for the sum of percentages is due to rounding errors.

DAX options and options on DAX futures".

The table 6.2 shows the stocks in the index with their market capitalization and closing prices at September 3, 1990 and December 30, 1987 respectively. We included a column that shows the percentage of every company as of September 1990. Using the equation given by Loistl (1991,p74) where we note the percentage of company i at time t by $\phi_{i,t}$, we have

$$\phi_{i,t} = \frac{100}{DAX_t}\left(\frac{p_{i,t}\cdot q_{i,t1}\cdot c_{i,t}\cdot K_{t1}}{\sum p_{i,t0}\, q_{i,t0}}\right)\cdot 1000\,.$$

Having said that our data basis consists of $p_{i,t}$ and $F_{i,t}$, we wanted an expression in $F_{i,t}$ rather than $c_{i,t}$. But if can rewrite the former equation as

$$\phi_{i,t} = \left(q_{i,t1}\cdot c_{i,t1}\cdot K_{t1}\cdot 100\cdot\frac{1}{\sum q_{i,t1}}\right)\sum q_{i,t0}\frac{p_{i,t}}{DAX_t}\cdot\frac{1}{p_{i,t0}\,q_{i,t0}}\cdot 1000\,,$$

where the expression in parentheses equals exactly what we defined earlier on as $F_{i,t}$. Hence

$$\phi_{i,t} = F_{i,t}\sum q_{i,t0}\frac{p_{i,t}}{DAX_t}\cdot\frac{1}{p_{i,t0}\,q_{i,t0}}\cdot 1000 \tag{6.4}$$

is the expression that we were looking for. The last column in table 6.2 is calculated using this equation.

We have discussed how the DAX formula takes into account the dividend payments and capital changes via the $c_{i,t}$. The last but one equation shows the dependence of the actual percentages of a company in the DAX, $\phi_{i,t}$, from these factors $c_{i,t}$ and the linking factors K_t. It is worth regarding how the factors evolved over time, we therefore present some descriptive statistics in table 6.3.

Table 6.3: Evolution of the $\phi_{i,t}$-values that give the percentage values of a share in the DAX

Symbol	Average	Min	MAX	Std.Dev.	Coef.Var.
ALV	0.121649	0.105167	0.135867	0.006512	0.053531
BAS	0.040768	0.036774	0.044386	0.001638	0.040182
BAY	0.050321	0.043034	0.056250	0.003799	0.075486
BHW	0.017896	0.016692	0.019364	0.000608	0.033974
BMW	0.024079	0.020529	0.028191	0.002411	0.100133
BVM	0.014394	0.013400	0.015656	0.000508	0.035288
CBK	0.019935	0.018173	0.021497	0.000732	0.036700
CON	0.006120	0.004635	0.008892	0.000972	0.158798
DAI	0.094829	0.082010	0.105506	0.006168	0.065039
DGS	0.007582	0.006744	0.008421	0.000300	0.039515
DBC	0.003446	0.003018	0.003847	0.000173	0.050140
DBK	0.087625	0.082514	0.092027	0.002236	0.025518
DRB	0.041792	0.037863	0.046327	0.002249	0.053809
HEN	0.010517	0.009224	0.012146	0.000684	0.065036
HFA	0.042328	0.035704	0.046744	0.002694	0.063652
KAR	0.014014	0.011953	0.016753	0.001077	0.076826
KFH	0.015031	0.013027	0.018370	0.001540	0.102438
LIN	0.015047	0.013872	0.016443	0.000624	0.041486
LHA	0.011708	0.009948	0.013150	0.000597	0.051001
MAN	0.018198	0.015729	0.020071	0.000890	0.048910
MMW	0.025134	0.022156	0.027775	0.001347	0.053578
MET	0.011664	0.009892	0.013164	0.000608	0.052094
PRS	0.013798	0.010857	0.015731	0.001254	0.090895
RWE	0.055460	0.049946	0.063409	0.003270	0.058956
SCH	0.015841	0.012505	0.017242	0.001018	0.064291
SIE	0.097676	0.091934	0.103640	0.003491	0.035740
THY	0.021205	0.019546	0.023065	0.000979	0.046145
VEB	0.045996	0.043967	0.048129	0.000948	0.020619
VIA	0.017732	0.016389	0.019203	0.000687	0.038751
VOW	0.038351	0.035397	0.046436	0.002734	0.071288

Source: Own calculations with the programme `phidemo`. We do not present this programme in the appendix as it is a straightforward combination of several calls to the function that calculates the ϕ_i values (cf. `daxdata.c`, section B.3) and a routine that calculates the descriptive values.

One sees clearly in table 6.3 that the portion of every company in the index varies over time. Taken as an example, the values for Allianz have a spread of over 3% between the highest and the lowest value in the value.

In the simple optimisation approach that we are undertaking in the next chapters, we are going to find the settings of parameters that come closest the these percentage values (as least when we are estimating with a portfolio of 30 shares).

6.5 Summary

In this chapter on the DAX, the Deutsche Aktienindex, we saw

- which different formulae were used or are still in use for stock market indices and that the majority of these is now calculated as a capitalization-weighted index,
- how the DAX is defined and how this formula can be simplified for daily calculations,
- how the concept of a total return index is realised using the factors $c_{i,t}$ and the chain linking factor K_t,
- which stocks form the DAX during the period that our data is based on and we visualized the DAX in a figure.

Chapter 7

Using Genetic Algorithms for Passive Management I

L'essentiel était de bien faire son metier.
ALBERT CAMUS (1947)

7.1 Introduction

Having introduced genetic algorithms as well as passive management (with the financial theory behind it — and the DAX for an application), we are now ready to try to bring together these two subjects. We are going to see whether the capacity of GAs to exploit vast parameter spaces can be used successfully to find index tracking portfolios with as low a tracking error as possible. We will start by considering the problem of minimizing the tracking error. Next we will try to calculate lower bounds for the tracking error. Thereafter, we will concentrate our efforts on two distinct experiments. For the first one, we force the GA to build a portfolio containing all thirty DAX shares. But we will let the GA find the weights for these thirty shares — so that we can compare his findings with the known solution that is given by the actual DAX portfolio. For the second experiment, we force the GA only to take thirty shares — so that it might decide to take a certain share two or three times, but the optimal solution is still known, it is the actual DAX portfolio where of course all thirty shares are selected. Roughly speaking, we give the GA the chance to construct a fully replicated index fund and we vary the initial condition with these two experiments.

7.2 Minimize the Variance

It is intuitively clear that one possible metric in order to quantify the "distance" between a market index and a portfolio is simply the variance of the times series resulting from taking the difference of the index return and the portfolio return. And, of course, minimizing the standard error instead of the variance is absolutely equivalent as the square root function is strictly monotonic. But there are also more formal arguments to it. Chamberlain and

Rothschild (1983) as well as Chamberlain (1983) consider in two papers arbitrage analysis on large asset markets — with an infinite number of assets. They define: *"One dollar invested in the ith asset gives a random return of x_i. A portfolio formed by investing α_i in the ith asset has a random return of $\sum_{i=1}^{N} \alpha_i x_i$; the portfolio is represented by the vector $(\alpha_1, \ldots, \alpha_N)$. Short sales are allowed, so α_i may be negative.*

There is an underlying probability space, and $L_2(P)$ denotes the collection of all random variables with finite variances defined on that space. The x_i are assumed to have finite variances, so that the sequence $\{x_i, i = 1, 2, \ldots\}$ is in $L_2(P)$. The means, variances, and covariances of the x_i are denoted by

$$\mu_i = E(x_i), \quad \sigma_{ii} = V(x_i), \quad \sigma_{ij} = \operatorname{cov}(x_i, x_j) .$$

We let $\mathcal{F}_N = [x_1, \ldots, x_N]$ denote the span of $x_1, \ldots, x_N$; i.e., the linear subspace consisting of all linear combinations of $x_1, \ldots, x_N$. Let $\mathcal{F} = \bigcup_{N=1}^{\infty} \mathcal{F}_N$, so that $p \in \mathcal{F}$ is the random return on a portfolio formed from some finite subset of the assets.

It is well-known that $L_2(P)$ is a Hilbert space under the mean-square inner product:

$$(p, q) = E(pq) = \operatorname{cov}(p, q) + E(p)E(q) ,$$

with the associated norm:

$$||p|| = \left(E(p^2)\right)^{1/2} = \left(V(p) + (E(p))^2\right)^{1/2} ,$$

for $p, q \in L_2(P)$." The second equality in the last but one equation is of course only a readjustment of the well-known statistical fact (cf. for example Bamberg and Bauer 1987, page 126)

$$\operatorname{cov}(x, y) = E(x \cdot y) - E(x) \cdot E(y) .$$

Chamberlain and Rothschild (1983) continue: *"Since $\mathcal{F}$ is a linear subspace of $L_2(P)$, its closure, $\overline{\mathcal{F}}$, is also a Hilbert Space. If $p \in \overline{\mathcal{F}}$, then there is a sequence $\{p_N\}$ in $\mathcal{F}$ with $E\left((p_N - p)^2\right) \to 0$ as $N \to \infty$. So there are finite portfolios whose random returns are arbitrarily good approximations to p."* So with (realisable) finite portfolios, we can approximately obtain the results for (unrealisable) infinite portfolios — as the market portfolio in Sharpe's (1964) capital asset pricing model, that is per assumption an infinite portfolio whereas the market index that is taken as a proxy for it must surely be a finite portfolio.

But we wanted to find out more about the use of the variance (or equally the standard error) for measuring our tracking portfolios. The linear subspace $\mathcal{F}$ of the Hilbert space $L_2(P)$ contains thus vectors of infinite size of asset returns that we can approximate with finite vectors p_N. And a vector v containing the difference between the returns of a market index and the returns of an index fund is also a vector from $\mathcal{F}$ — so we have good arguments at hand to use its norm $||v|| = \left(E(v^2)\right)^{1/2}$ or, especially, its variance.

We will now try to make precise mathematically what kind of expression we can derive from the variance approach. Our goal is to replicate an index over some time period. The 'optimal' portfolio will have a certain return structure during that period that we describe by the vector of returns; we do the same for the index returns. It is therefore rather straightforward to take the difference of these returns and then to measure the variance of this return difference as to quantify the 'dispersion' of a time series. In mathematical notation:

$$\min_{x_i\ \forall i \in P} \operatorname{var}\left(r_{p,t} - r_{m,t}\right)\ \ \forall t \in T \tag{7.1}$$

where

$r_{p,t}$	$= \sum_{i\in P} x_i r_{i,t}\ \ \forall t \in T$		
$r_{m,t}$	$= \sum_{i\in I} \phi_i r_{i,t}\ \ \forall t \in T$		
$r_{i,t}$	= return of share i at time t		
x_i	= proportion of share i in portfolio, constant over t		
ϕ_i	= proportion of share i in index, constant over t		
I	$= 1, 2, \ldots, 30$	(for the DAX index of 30 shares)	
P	$= \mathcal{P}(I)$	(any set constructed by permuting I and possibly omitting some of its members)	
T	$= t_{(1)}, t_{(2)}, \ldots, t_{(\omega)}$	(where ω might be 50 or 100 or any other reasonable figure that is fixed for the optimisation)	

or equally

$$\min_{x_i\ \forall i \in P} \operatorname{var}\left(\sum_{i\in P} x_i r_{i,t} - \sum_{i \in I} \phi_i r_{i,t}\right)\ \ \forall t \in T\ .$$

Note that this notation stresses two things. First, when we write $r_{i,t}$ we say clearly that returns vary by share and by moment in time (days for our database). Second, x_i and ϕ_i do not possess the index t — their are meant to be constant over the training period T for the index fund. We let x_i be constant for our GA optimisation approach — but this could surely be overcome by a more sophisticated approach. Still, it is most often a good idea to start from the beginning, i.e. from the simplest situation. That ϕ_i is held fixed follows from the development that we are about to begin. This is certainly an obstruction on the way to a tracking error of zero; we have discussed in the last chapter the changes in the ϕ_is during the optimisation period. So fixing the ϕ_i will unfortunately cause a residual tracking error.

We will now redefine the index set for P by setting $x_i = 0$ if $i \in I \wedge i \notin P$ and write more compactly

$$\min_{x_i\ \forall i \in I} \operatorname{var}\left(\sum_{t\in T}\sum_{i\in I} (x_i - \phi_i) r_{i,t}\right)\ .$$

Next, we will drop the time index t for notational convenience (but we still imply the context described above), so the last equations becomes

$$\min_{x_i\ \forall i \in I} \operatorname{var}\left(\sum_{i\in I} (x_i - \phi_i) r_i\right)\ .$$

We will demonstrate for dimension 2 that

$$\operatorname{var}\sum_{i=1,2}(x_i - \phi_i) r_i = \sum_{i=1,2}\sum_{j=1,2}(x_i-\phi_i)(x_j-\phi_j)\operatorname{cov}(r_i, r_j)\ .$$

We start with

$$\operatorname{var}\sum_{i=1,2}(x_i - \phi_i) r_i = \operatorname{var}\left((x_1-\phi_1)r_1 + (x_2-\phi_2)r_2\right)$$

where we apply $\operatorname{var}(\alpha x + \beta y) = \alpha^2 \operatorname{var}(x) + \beta^2 \operatorname{var}(y) + 2\alpha\beta \operatorname{cov}(x,y)$, this yields

$$\operatorname{var}\sum_{i=1,2}(x_i - \phi_i) r_i = (x_1-\phi_1)^2\operatorname{var}(r_1) + (x_2-\phi_2)^2\operatorname{var}(r_2) + 2(x_1-\phi_1)(x_2-\phi_2)\operatorname{cov}(r_1,r_2)$$

and with $\operatorname{var}(x) = \operatorname{cov}(x,x)$, we finally obtain

$$\operatorname{var}\sum_{i=1,2}(x_i - \phi_i) r_i = \sum_{i=1,2}\sum_{j=1,2}(x_i-\phi_i)(x_j-\phi_j)\operatorname{cov}(r_i, r_j)$$

which completes the proof. As this is equally true for dimensions higher than two (though it is more laborious to prove), we will use it for the minimization problem. Hence we get

$$\min_{x_i\ \forall i \in I}\sum_{i\in I}\sum_{j \in I}(x_i-\phi_i)(x_j-\phi_j)\operatorname{cov}\left(r_i, r_j\right)$$

which itself is

$$\min_{x_i\ \forall i \in I}\left[\sum_{i\in I}\sum_{j\in I} x_i x_j \operatorname{cov}(r_i,r_j) - 2\sum_{i\in I}\sum_{j\in I} x_i\phi_j\operatorname{cov}(r_i,r_j) + \sum_{i\in I}\sum_{j\in I}\phi_i\phi_j\operatorname{cov}(r_i,r_j)\right]\ .$$

Or in matrix notation, we get

$$\min_{x_i\ \forall i \in P}\left(x^\top C x + \phi^\top C\phi + x^\top p\right) \tag{7.2}$$

where

$$\begin{aligned} C &= \left(\operatorname{cov}(r_i,r_j)\right)_{i,j\in I} \\ p &= \left(-2\sum_{j\in I}\phi_j\operatorname{cov}(r_i,r_j)\right)_{i\in I} \\ x &= (x_i)_{i\in I} \\ \phi &= (\phi_i)_{i\in I}\ . \end{aligned}$$

Noting that $x_i = 0$ if $i \in I \wedge i \notin P$, we can write

$$\min_{x_i\ \forall i \in P}\left[\sum_{i\in P}\sum_{j\in P} x_i x_j \operatorname{cov}(r_i,r_j) - 2\sum_{i\in P}\sum_{j\in I} x_i\phi_j\operatorname{cov}(r_i,r_j) + \sum_{i\in I}\sum_{j\in I}\phi_i\phi_j\operatorname{cov}(r_i,r_j)\right]\ ,$$

where we can drop the last term. This facilitates the real calculation as this last part is constant in x_i; we nevertheless have to add its value to the final expression of the numerical value of the variance for any choice of the x_i. The advantage of the last expression is that for portfolio of m shares as compared to n in the index, the calculations become a little faster if m is smaller than n. In appendix B.2, we give the file `eval.c` with the actual coding of the calculations.

Having established this approach of minimizing the variance, we are now ready to define our tracking error that we use through-out the application part.

Definition 7.1 *The* tracking error *that we use is the usual standard error of the difference of the returns of the index and of the index tracking portfolio:*

$$\mathrm{TE} = \left(\frac{1}{T}\sum_{t=1}^{T}\left((r_{p,t} - r_{m,t}) - E(r_{p,t} - r_{m,t})\right)^2\right)^{1/2}\ ,$$

where the index p stands for the index tracking portfolio, the index m for the market index.

This definition differs from the Meade and Salkin (1990) tracking error (see equation (5.1)) as it takes into account explicitly the expected value of this return difference. This last expected value might be zero at the end of the optimisation process, but certainly not during the course. But Meade and Salkin (1990) do not claim that equation (5.1) gives the standard error, they called their value appropriately 'root mean squared error' or r.m.s.e. for short. Note finally that we calculated our returns as

$$r_{i,t} = 100 \cdot \frac{p_{i,t} - p_{i,t-1}}{p_{i,t-1}}$$

where $p_{i,t}$ is the price of share i at time i. This means that, say, a ten per cent increase is noted as $r_{i,t} = 10$ and not as 0.1. This last value would signify a return of a tenth of a per cent. So a tracking error of 0.5% would mean that in about 68% of the time, the return of the index fund is as close as 0.5% to the return on the market index. We hereby assume that the returns are normally distributed.

7.3 Establishing some Bounds for the Tracking Error

After having provided our tracking error definition, we will now consider some aspects and factors that influence the calculated value of a tracking error. You simply cannot expect to get a zero tracking error because the index is a mathematical construction (which reality is not very often). Consider the following examples. Take as a first example the time-lag for a share that goes ex-dividend and the actual payment of the dividend. What you can do is either take a credit to buy the shares earlier but you then have to account for the credit costs or wait until the payment arises but you then miss some earnings for the money not yet invested. Second example: New shares that are offered at a certain price but that you might have to buy in the market where there is an immediate premium on top of the official offering price. That means that your returns will be lower. Tomlinson (1990) states as the classic example the issue of British Telecommunications where the shares were issued at 50p whereas the market price was around 90p. Third example: Trading costs that a real-world index fund has to pay (and hence to earn) whereas the index as a mere mathematical construction is free of trading charges. Tomlinson (1990) gives more examples: Whereas the index is by definition always fully invested, the index fund will inevitably carry some cash. Purchases can only be made in permissible lot sizes whereas the index does not suffer these difficulties. Stocks that are suspended from trading are generally removed from indices, while they continue to diminish portfolio returns. Tomlinson, after having evoked the time-lag problem as well as the taxation problem that arise with dividends, concludes: *"The returns on most indices are in fact generally calculated on a capital basis, with the effects of dividends treated separately if at all. Calculation of total returns is then a separate exercise which involves some subjective judgement. It is interesting that there are frequently quite significant differences between total returns on the same indices as calculated by different organizations. These differences are often comparable in magnitude with the tracking error that an index fund may experience.*

The message is simple — those considering indexation must be aware that managing index funds is a highly technical exercise and that it may in fact not be possible to exactly replicate the performance of a target index, or even to agree categorically on what that performance has been."

Concretely for the DAX that we try to track: you cannot expect to get a tracking error of zero for the DAX when there are dividend payments or rights issues during the tracking period. When we discussed equation (6.1) for the DAX, we said that such an event is taken into account by a change in the factor c_i for the share i that pays dividends or issues rights. But that means that, from that day onwards, the factors ϕ_i that give the percentage of share i in the DAX are changed for the calculation of the DAX. We showed in table 6.3 on page 66 the change that arises in these factors. So even if the index fund is constructed by taking all of the true ϕ_i-values (either the ones that were valid before the event or the ones that are valid after it), the index fund will be exposed to some tracking error. We thus have to fix the ϕ_i values for a certain date. The most appropriate day to do so is the last day of the tracking period. The reason for doing so is that the normal setting for an optimisation run would be at a certain instant t_0 with last information from day t_{-1}. The goal of the optimisation is to minimize from t_0 into the (uncertain) future. Clearly, the best bet for an index fund of 30 shares is full replication or to adopt the ϕ_i values directly. To illustrate this, we formed two index fund portfolios with fixed coefficients ϕ_i for a period of 100 trading days (observations) from September 3, 1990 until January 9, 1991. We chose two different set of coefficients of that portfolio, one from the start of the period and the other at from its end. If we fix the ϕ_i for the last day of the trading period, we get TE = 0.0004 whereas with the coefficients fixed for the first day of the period, we obtain TE = 0.0196. If we generously account for rounding errors for the first value, we cannot argue about the second. This is the 'normal setting' that we described above: we fix as good as we can at the beginning, but changes in the DAX stock account for about 0.02% of the tracking error. And this is a period with little change in the factors c_i. We will refer to this tracking error level as *residual tracking error*.

So how much (or how little) of a tracking error achieve the existing index funds? Liesching and Manchanda (1990) give some figures for the annual tracking error for optimised sampling of the FTA-All Shares index (that comprises about 683 stocks) and that we reprint in table 7.1. They state: *"In general, a well designed index fund in a major market should have a predicted tracking error of approximately 0.5%, or lower."* One notes immediately that "the bigger, the better" holds another time, but that is hardly surprising as strategies called "naive diversification" (by a large number of stocks to minimize systematic risk; see chapter 4) are well known in Finance (Göppl 1990).

Table 7.1: Reported Tracking Errors for optimised sampling of the FT-A All Share Index

Tracking Error	Number of stocks
0.30%	400
0.35%	350
0.50%	275
0.60%	200
1.00%	150
3.00%	30

Source: Liesching and Manchanda (1990)

We have at our disposal a Deutsche Bank internal memorandum from 1990 that sums up on three pages indexation methods (these arguments are nearly identical to the ones by Liesching and Manchanda (1990) that we discussed) as well as results already achieved by financial firms:

- for the USA, a fully replicated fund tracked the S&P 500 with a mean tracking error of approximately 0.2% since 1972,
- for the UK, a fully replicated fund that achieved a tracking error of 0.2% over the last six years (i.e. 1983–1989) and
- a fund that uses optimised sampling that met its goal of 0.5% during the last five years.

The author concludes: *"1. One can reasonably expect to achieve a mean tracking error of 0.2% only with replication, not with either of the sampling techniques. 2. A reasonable mean tracking error to set as a goal for either of the two sampling techniques is 0.5%. 3. Some passively-managed funds have goals for mean-tracking error as high as 1% and apparently can stay in business with this rate."*

So we now have our bounds. If we do not achieve at least 0.5%, the result can be called disappointing as even stratified sampling and full replication do better. Anything between 0.2% and 0.5% would be acceptable, but not glorious. Finally, if we can beat 0.2%, we might claim that the genetic algorithm approach is worth trying.

7.4 Representation for GENESIS

We present GENESIS in appendix A where we moreover show the simplest example of an evaluation function from the file `f1.c`. The question that us interests here is as how we can use the same interface to the GENESIS system for our passive management application. GENESIS is supposed to be used as a 'black box' system; the user is expected to provide a problem specific function named `eval.c` that is equipped with the function interface as defined by GENESIS. This idea of a robust mechanism that performs the given task without problem specific knowledge is at the heart of the GA idea; we have discussed this point already in the first chapter. Fortunately, GENESIS allows a rather high-level representation of the problem. The user has to choose between a bitstring representation or a floating point representation. This latter is of course very comfortable for a numerical optimisation problem as the one that we are considering here. Having chosen this representation, the user has to specify the number of elements in the vector `vect[]`; either with the interactive programme `setup` or by directly editing the file `template`. These elements in `vect[]` do not have to be of the same form.

For our problem, we have chosen to place two different types of elements in the vector. The first type is for selecting a share out of the thirty different shares that the DAX comprises. We thus refer to these values as *selectors*. We therefore chose an integer representation of the set $0, 1, 2, \ldots, 31$ (note that GENESIS requires for the range a power of 2 — binary coding!). This means that one in sixteen values will not be valid as there is no corresponding share in the DAX. We generously take this into account and flag these values as "no share selected". It is clear that a portfolio with a bigger number of shares has greater chances to be broadly diversified and thus closer to the index as a smaller portfolio. There is hence a kind of automatic selection away from these values — portfolios with selectors that actually select no share at all are very likely to have a lower fitness value. Or equally, they might even be ameliorated in one of the next generations when this invalid selector value might be overwrittem by a valid value.

The second type in the vector is for determining the actual amount of shares to be bought from the share chosen by the corresponding selector. We called these values hence *weights*. The relation between selectors and weights is straightforward. We take as many selectors as weights, the number n of elements is hence even. The first $n/2$ vector elements are selectors and the remaining $n/2$ are weights where there is a clear relation between them. Each selector can be indexed by i where $i = 1, \ldots, n$ and the corresponding weight is indexed by $i+n/2$. The range of possible values for the weights is usually chosen as the unit interval with 1024 possible values. This might seem restrictive, only a tenth of a per cent is distinguishing neighbouring values, but we tried higher values as 8096 or even 32768 which seemed not to have any impact on the quality of constructed index tracking portfolio. The actual selector value does not directly indicate the percentage of the corresponding share. As there is absolutely no constraint for forcing the sum of the weights to equal 1, we normalize in the function `eval` by dividing each of the weight values by the sum of all weight values in the vector.

The representation offers a good degree of flexibility. A next step could be to offer the choice of preselecting either certain shares or even certain shares with a given weight. This would allow to force some shares with a known high correlation with the index to be included. Equally, shares with a high market capitalization could be forced into the index. Recall that including such a constraint considerably reduced the tracking error in Meade and Salkin's (1990) second paper. Another possibility that this next step would offer is to include existing portfolios.

7.5 Regressing a Tracking Portfolio

We will now establish a benchmark for the genetic algorithm runs by simply estimating a tracking portfolio by using ordinary least squares (OLS) estimation. The idea is straightforward: the GA does use historic stock exchange data to estimate portfolios, so we can formulate an econometric model for the DAX as the following:

$$DAX_t = \beta_1 ALV_t + \beta_2 BAS_t + \ldots + \beta_{29} VIA_t + \beta_{30} VOW_t + \varepsilon_t$$

where the index t symbolize that we are regressing on vectors, i.e. we regress the DAX on the daily data for its thirty constituents. We changed the standard econometric notation; instead of writing x_i, we write the three-letter-shortcut for each stock that is in use at the Frankfurt Stock Exchange that we have introduced in table 6.2. The vector for the DAX contains the daily returns and the vectors for the DAX-companies contain the daily capitalization-weighted returns, i.e. the returns obtained from the 'price times capital'-products and not from the prices. This follows the DAX formulae as stated in chapter 6 where we of course do not account for total-return adjustments. The idea behind the regression is to get the best estimation that can be held constant over the training period — so no adjustment will be made in case of dividend payments.

We now give the results for the first estimation for a in-sample training period of 100 days as for the GA-setting where the t-test values are given in parentheses under the corresponding parameter estimates:

$$DAX_t = \underset{(29.80)}{0.1206}\, ALV_t + \underset{(5.75)}{0.0356}\, BAS_t + \underset{(6.76)}{0.0448}\, BAY_t + \underset{(6.87)}{0.0296}\, BHW_t$$

$$
\begin{aligned}
&- \underset{(-0.90)}{0.0092}\, BMW_t + \underset{(2.11)}{0.0093}\, BVM_t + \underset{(3.64)}{0.0193}\, CBK_t - \underset{(-1.74)}{0.0068}\, CON_t \\
&+ \underset{(12.13)}{0.1118}\, DAI_t + \underset{(5.54)}{0.0671}\, DGS_t - \underset{(-0.08)}{0.0002}\, DBC_t + \underset{(9.64)}{0.0894}\, DBK_t \\
&+ \underset{(9.04)}{0.0449}\, DRB_t + \underset{(0.52)}{0.0069}\, HEN_t + \underset{(8.54)}{0.0464}\, HFA_t + \underset{(2.50)}{0.0280}\, KAR_t \\
&- \underset{(-9.19)}{0.0933}\, KFH_t + \underset{(1.93)}{0.0113}\, LIN_t + \underset{(3.31)}{0.0072}\, LHA_t + \underset{(4.76)}{0.0195}\, MAN_t \\
&+ \underset{(4.79)}{0.0212}\, MMW_t + \underset{(0.39)}{0.0018}\, MET_t + \underset{(5.39)}{0.0157}\, PRS_t - \underset{(-4.83)}{0.0593}\, RWE_t \\
&+ \underset{(0.43)}{0.0061}\, SCH_t + \underset{(10.35)}{0.0994}\, SIE_t + \underset{(3.33)}{0.0229}\, THY_t + \underset{(0.43)}{0.0053}\, VEB_t \\
&+ \underset{(10.21)}{0.1095}\, VIA_t + \underset{(4.51)}{0.0377}\, VOW_t + \varepsilon_t\ .
\end{aligned}
$$

The sum of squared residuals equalled 0.1206, the explained sum of squares 371.5108; the null hypothesis of the non-significance of all parameters can be rejected at virtually any significance level ($F = 7151.4$). Still, the regression has given some strange results. Some shares have negative coefficients, so that one is tempted to assume a negative correlation between these shares and the DAX during the sample period and some shares have quite non-significant parameters, we will simply leave them out for the subsequent regression. More interesting are the actual parameter values. For Allianz (ALV), the value of 0.1206 corresponds to the percentage that this company has in the DAX, similar relations can be found for a good number of shares as one can compare with table 6.3. On the other hand, it is quite puzzling to find for example Viag (VIA) with a parameter estimate of 0.1095 which is about seven to eight times its actual percentage in the DAX.

In a second attempt, we tried to regress the DAX on all those shares who had a value of $t > 1.67$, i.e. we excluded all non-significant variables and variables with negative estimates as we are not allowing short-sales (i.e. negative positions in shares) in this simple setup. The constraint of having all parameters with a t-value higher than 1.67 led to two more regressions because there was one variable in each of the last two regressions that fell into the rejection regions after the change in the regression model; we give now the "final" model along with its estimates.

$$
\begin{aligned}
DAX_t \;=\; &\underset{(17.23)}{0.1089}\, ALV_t + \underset{(4.24)}{0.0430}\, BAS_t + \underset{(6.71)}{0.0683}\, BAY_t + \underset{(4.25)}{0.0291}\, BHW_t \\
&+ \underset{(2.26)}{0.0190}\, CBK_t + \underset{(6.80)}{0.0428}\, DAI_t + \underset{(2.84)}{0.0134}\, DGS_t + \underset{(10.00)}{0.1295}\, DBK_t \\
&+ \underset{(4.89)}{0.0415}\, DRB_t + \underset{(4.03)}{0.0355}\, HFA_t + \underset{(2.89)}{0.0203}\, KAR_t + \underset{(2.19)}{0.0185}\, LIN_t \\
&+ \underset{(2.55)}{0.0069}\, LHA_t + \underset{(3.11)}{0.0210}\, MAN_t + \underset{(5.71)}{0.0388}\, MMW_t + \underset{(5.01)}{0.0130}\, PRS_t \\
&+ \underset{(8.29)}{0.1216}\, SIE_t + \underset{(7.76)}{0.0363}\, VIA_t + \underset{(8.59)}{0.0446}\, VOW_t + \varepsilon_t\ .
\end{aligned}
$$

The sum of squared residuals equalled 0.4305, the explained sum of squares 371.2009; the null hypothesis of the non-significance of all parameters can still be generously rejected ($F = 3658.2$). It is now time to evaluate the portfolio estimated by OLS on the data. Table 7.2

shows the composition in the format that we are going to use for the portfolios obtained by the genetic algorithm. It is interesting to compare these values to the proportions taken up in the DAX, see tables 6.2 and 6.3. We are interested in how well (or badly) this portfolio

Table 7.2: Composition of the portfolio estimated by OLS

```
# Portfolio from min-1.ol2
# from generation 1, evaluation 1 with value 0.007000
# Setup: 19 selectors, 19 weights, 19 effectively selected
ALV  0  12.783942   BAS  1   5.045008   BAY  2   8.011541   BHW  3   3.410457
CBK  6   2.232481   DAI  8   5.020797   DGS  9   1.582550   DBK 11  15.201494
DRB 12   4.874114   HFA 14   4.162570   KAR 15   2.382193   LIN 17   2.166949
LHA 18   0.815005   MAN 19   2.469705   MMW 20   4.554611   PRS 22   1.526395
SIE 25  14.264979   VIA 28   4.256960   VOW 29   5.238247
```

Source: Own computation with the programme `showpf` (cf. appendix B.6).
Note: The values in line two (generation number, evaluation number, evaluation value) are faked as this portfolio was not estimated by the GA; we simply needed some values to be in line with the data format required by `showpf`.

behaved during the training period, the first 100 observations, i.e. we are interested in the TE-value according to definition 7.1. But that is not the only figure of merit. In a real-world setting, i.e. with "real" money to be invested in an index fund, the more interesting question is not how the portfolio behaved in-sample but how it scored ***out-of-sample*** as this gives an idea of how good the approximation really is. Imagine that at a certain date, we have at our disposal historic stock market data, say for hundred days as in this setting. So if we refer to the actual day that the optimisation for the best index tracking portfolio is carried out as t_0, our data lasts from t_{-1} to t_{-100} (it is early morning at t_0, so we have no data yet for the day). The tracking error according to definition 7.1 gives one figure in order to describe how the tracking portfolio was fitted on the tracked index. In the real-world setting, one would be exposed to tracking error from t_0 onwards for the time to come, i.e. the dispersion between the return pattern of the tracking portfolio and the actual index as given day by day. We cannot use the same tracking error definition to describe this dispersion. Why? Recall from chapter 6 that we fixed the proportion of every share in the DAX (we noted this value with ϕ_i for share i) in order to allow the matrix notation of the problem. We equally discussed the question of the non-constancy of these factors over time — they reflect the part that the product of capital times share price times correction factor takes up in the DAX at a time. Obviously, these values change in magnitude so that the ϕ_i vary as well. Besides, this is precisely why OLS estimation does not give a perfect fit. So the tracking error 7.1 that is used as the objective function value for the genetic algorithm is calculated with these fixed values. For the tracking error of the out-of-sample periods, we use the standard error between the actual index returns and the returns on the portfolio.

Definition 7.2 *The* real tracking error *that we use for the out-of-sample period is the usual standard error of the difference of the returns of the index as given at the stock market and of the index tracking portfolio:*

$$
\mathrm{RTE} = \left(\frac{1}{T} \sum_{t=1}^{T} \left((r_{p,t} - r_{m,t}) - E(r_{p,t} - r_{m,t}) \right)^2 \right)^{1/2} ,
$$

where the index p stands for the index tracking portfolio, the index m for the market index.

This figure gives the dispersion that one really is exposed to when holding the index tracking portfolio. Table 7.3 gives these values for the three out-of-sample periods that we could construct from our data. As we used 100 days for training, we suppose to use 50 days as the period length for the life of the fund. We hence get a period P1 for the first hundred days, the in-sample-period, and periods P2 and P3 with 50 days each and finally period P4 with the remaining 46 days. The surprisingly high value for period P1 can easily be explained by the

Table 7.3: Tracking Errors for the OLS estimated portfolio

TE P1	RTE P2	RTE P3	RTE P4
0.10752	0.26386	0.19360	0.27922

Source: Own computations with the programme `analyse` (cf. appendix B.5).
Note: For this table as well as for the one's that follow, the unit of measurement for the tracking error is per cent.

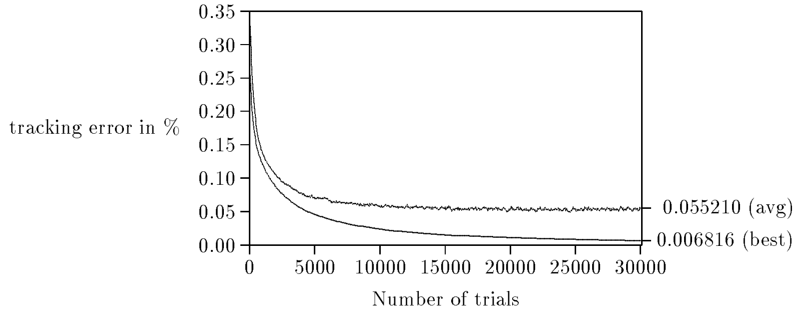
fact the index values used for the optimisation equal an artificial index in the sense that the ϕ_i are held fixed — whereas the tracking error RTE compares to the real index values. This evidently gives a higher value. But our aim is to have low values for the periods P2 to P4; the values obtained compare quite favourably to the Deutsche Bank estimates of a tracking error of 0.2 for replicated funds and of 0.5 for stratified or optimised sampling. So we now have at our disposal some reference tracking errors to which we will compare the tracking errors of the GA estimated portfolios.

7.6 Verifying the Convergence

As a first application of the genetic algorithm for the indexation problem, we now are going to study the genetic algorithms performance for a problem with a known solution. We will let the GA solve the index fund construction problem for an index fund size of 30, i.e. with every share included. The known solution is then of course full replication; the GA shall find as his optimal weights the coefficients ϕ_i in the DAX. For the simplest possible setting, we fix the 30 selectors with their position number (e.g. the fifth selector was given to the value 5) and let the GA determine the optimal weight for each share. Figure 7.1 shows the evolution of the values "best-of-generation" and "generation average" over the GA run; recall that GAs work with populations, so we plot the best value within the population as well as the population average for every population.

The values in figure 7.1 are thus very satisfying considering the bounds presented in 7.3. The average of the best individuals in the final generation of each of the 100 runs is $6.816 \cdot 10^{-3} \approx 68/10000$ with a variance of the values of $3.944 \cdot 10^{-6}$. As GA results are achieved by a kind of 'guided random search', they depend on the outcomes of the pseudo-random number generator. The values produced by the latter vary by definition so that the GA results vary as well. So one repeats the individual experiment several times, here one hundred times. We fixed the number of best individuals that were saved at the end of an experiment to 10. As each experiment gives already ten individuals that represent for us ten portfolios, we obtain by repeating one hundred times exactly thousand individuals or portfolios. How are we judging on these one thousand portfolios? We sort the portfolios by the reported objective

Figure 7.1: Tracking error as the best-of-generation and the generation average over the number of evaluations for the first convergence experiment



Source: The data was calculated by the GENESIS programme `report`.
Note: The parameters for the GENESIS run were: population size 20, string length 150, crossover probability 0.95, mutation probability 0.005, gray coding, elitist strategy, two-point crossover. This run was repeated 100 times with different seeds for the random-number generator; total time consumption was about 15 hours on a SUN SPARC workstation.

value, the tracking error TE, and compare then its out-of-sample tracking errors RTE with the best ones reported for these values. The best evaluation was obtained in replication 57, the evaluation of individual 29894 in generation 1708 was $2.972 \cdot 10^{-3}$ (recall that we limited the experiments to 30000 evaluations for each replication). This is considerably better than the average of all best individuals which was $6.816 \cdot 10^{-3}$. We show this portfolio in table 7.4. The reported values compare quite favourably to the percentages of each share in the DAX

Table 7.4: Composition of the best portfolio estimated by the GA for the first convergence experiment

```
# Portfolio from min-57.c00
# from generation 1708, evaluation 29894 with value 0.002972
# Setup: 30 selectors, 30 weights, 30 effectively selected
ALV  0  12.733002   BAS  1   3.891348   BAY  2   5.262889   BHW  3   1.684125
BMW  4   2.070708   BVM  5   1.409816   CBK  6   2.094949   CON  7   0.686408
DAI  8   8.405313   DGS  9   0.798683   DBC 10   0.299825   DBK 11   9.240996
DRB 12   4.152898   HEN 13   0.984958   HFA 14   4.390207   KAR 15   1.409816
KFH 16   1.446816   LIN 17   1.434058   LHA 18   1.060233   MAN 19   1.846158
MMW 20   2.731599   MET 21   0.984958   PRS 22   1.122750   RWE 23   5.612473
SCH 24   1.645849   SIE 25  10.326746   THY 26   1.883157   VEB 27   4.789548
VIA 28   1.684125   VOW 29   3.915590
```

Source: Own computation with the programme `showpf` (cf. appendix B.6).

as stated in table 6.3. We next regard the tracking errors for this portfolio. We obtained the portfolio under consideration by sorting for the best TE value — we use the same method to find values to compare to by sorting for the best values of RTE in periods P2, P3 and P4, i.e. the "real" tracking error against the index in the out-sample-periods. Table 7.5 shows in the first line the values for the best portfolio for TE in period P1 and give in the next three lines the best portfolios for RTE P2, RTE P3 and RTE P4. There are several conclusion

Table 7.5: Tracking Errors for the first convergence experiment

run	gen	eval	TE P1	RTE P2	RTE P3	RTE P4
57	1708	29894	0.0029719	0.2559949	0.2095806	0.2651610
51	1698	29773	0.0129870	0.2442437	0.2034271	0.2515597
51	1698	29773	0.0129870	0.2442437	0.2034271	0.2515597
51	1690	29636	0.0130200	0.2442736	0.2034612	0.2515475

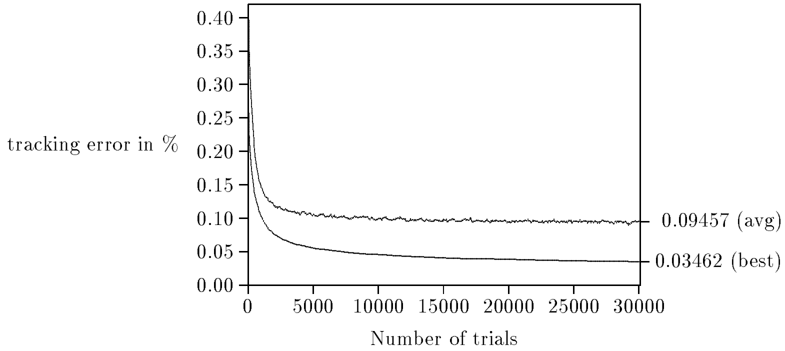
Source: Own computations with the programme `analyse` (cf. appendix B.5).

that can be made immediately. First, the best in-sample-period result does not automatically provide the best out-of-sample-periods results; the values are nevertheless quite close to each other. Second and inversely, the best out-of-sample-period results come with a quite bad in-sample-period result; the value is more than four times as high. Third, one portfolio won two contests — the best performer for RTE P2 also was best for RTE P3. Fourth, compared to the OLS-estimated portfolio whose tracking errors are reported in table 7.3, the results are not clear cut: the GA won two out of three comparisons but lost one (RTE P3).

We now try the same experiment with a slightly harder setting: instead of forcing each of the 30 DAX shares into the index, we will let GENESIS choose 30 selectors as well as 30 weights. So we let the GA decide whether to use thirty different shares or to replace some shares by selecting a second or third time a company already present. There is a reason for this: recall that our function interface provide a vector comprising selectors and weights. Each weight can be chosen between 0 and 1 and is later normalized by dividing by the sum of all weights. The possible values are hence likely to be quite comparable in size whereas in the DAX, the shares of Allianz, Daimler-Benz, Deutsche Bank and Siemens have a bigger part than all the other shares. So our representation allows to replace a "minor" company by a second stake in a "major" company. Figure 7.2 gives again the evolution of best and average tracking error over the GA runs.

The first result is obvious — the GA had more work to do by selecting shares and finding good weights for these shares, so the obtained value for the average of the tracking error of all reported best individuals over the 100 replications is $9.707 \cdot 10^{-3}$ with variance $1.34 \cdot 10^{-6}$ and thus about five times as high as the corresponding average for the average of the best individuals from the first experiment. The best value was obtained in run 92 within generation 1640, evaluation 29723 with a value of $2.291 \cdot 10^{-2}$. We show the portfolio that represents this individual in table 7.6. The genetic algorithm found hence his best portfolio by selecting 25 different companies. But among the suppressed companies is for example Karstadt (with symbol KAR and number 15) that in the OLS approach was part of all models. The other four suppressed titles were also suppressed by the OLS. The tracking errors for this portfolio as well as for the best portfolios for RTE P2, RTE P3 and RTE P4 are given in table 7.7. We can make the same comments as for the tracking errors of the first convergence experiment: good in-sample performance is not automatically related to good out-of-sample performance. Furthermore, there seem to be many alternatives for the tracking portfolios. We here looked for the best one for four different categories — and found four different portfolios. It is also worth noting that the winner for the RTE P3 scored better than the corresponding winner for the first experiment, but not as good as the simple OLS estimation. On the other hand, all three winners for the best RTE for periods P2 to P4 outperformed their colleagues from

Figure 7.2: Tracking error as the best-of-generation and the generation average over the number of evaluations for the second convergence experiment



Source: The data was calculated by the GENESIS programme `report`.
Note: All parameters as in given in figure 7.1. Calculation time was nearly 16 hours on the SPARCstation.

Table 7.6: Composition of the best portfolio estimated by the GA for the second convergence experiment

```
# Portfolio from min-92.c01
# from generation 1640, evaluation 29723 with value 0.022914
# Setup: 30 selectors, 30 weights, 25 effectively selected
ALV  0  12.582558   BAS  1   3.447749   BAY  2   5.270051   BHW  3   2.167800
BMW  4   2.702116   BVM  5   0.361568   CBK  6   2.513298   DAI  8   7.783349
DGS  9   1.759630   DBK 11   8.835913   DRB 12   4.139549   HEN 13   0.628324
HFA 14   4.689132   KFH 16   1.939610   LIN 17   1.876938   LHA 18   1.264684
MAN 19   1.680888   MMW 20   3.573896   PRS 22   1.154606   RWE 23   5.411464
SCH 24   2.575969   SIE 25  11.852191   VEB 27   5.403429   VIA 28   1.774896
VOW 29   4.610391
```

Source: Own computation with the programme `showpf` (cf. appendix B.6).

the first experiment and from the OLS estimation.

Still, the question to be answered in this section was whether the genetic algorithm can be said to converge, i.e. can we hope to annihilate the tracking error after a "many" evaluations, "many" genetic algorithm generation, "much" computer time. We found two results here:

Table 7.7: Tracking Errors for the second convergence experiment

run	gen	eval	TE P1	RTE P2	RTE P3	RTE P4
92	1640	29723	0.0229140	0.2506461	0.2120041	0.2595838
29	1631	29472	0.0297500	0.2377169	0.2037345	0.2476865
22	1651	29920	0.0420150	0.2378143	0.1980174	0.2494920
87	1641	29768	0.0294580	0.2392768	0.2026608	0.2450963

Source: Own computations with the programme `analyse` (cf. appendix B.5).

the first one for a setting where only the weights of the shares had to be found. Here the dimension of the remaining tracking error is satisfying — especially in the light of what we said about the residual tracking error earlier on in section 7.3. The second result is not that clear-cut. The remaining tracking error after 30000 evaluations was higher than for the first experiment but still about the size of what we qualified as the residual tracking error. Moreover, the out-of-sample tracking errors were not that fundamentally different. So we dare to answer the question of the convergence with a prudent “yes, possibly”.

7.7 Finding the Optimal Tracking Portfolio Size

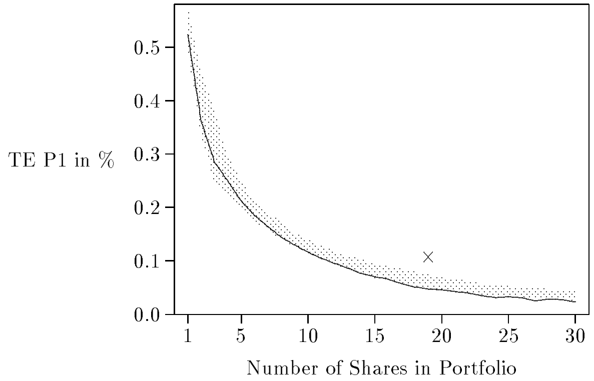
After having studied the capacity of the genetic algorithm to solve the index fund construction problem, we will now examine how portfolio size and tracking error are related. We conducted a rather big experiment in repeating the set-up used for the second convergence experiment for all other possible portfolio sizes. For the DAX, this meant to try all portfolio sizes between one share and twenty-nine shares (as thirty was already tested in the second convergence experiment). For each size, we repeated 100 experiments which each involved the creation of 30000 GA-individuals. The total computer time consumption was about ten days on a SUN IPC SPARCstation. We created approximately 150 Megabyte of data of which we kept about 12 Megabyte (each of the experiments contained 100 runs that were executed until 30000 individuals were evaluated, the report interval was set to 50 so that every 50 evaluations a convergence statistic was written to the `out`-file; all this yields $100 * 30000/50 = 60000$ lines in one `out`-file which gives about 4.3 Megabyte; the `out`-file is then read by `report` producing the report-file so that we could erase the `out-file` after an experiment).

The first question to be examined was of course how the objective function, the tracking error TE, evolved with different portfolio sizes. In figure 7.3, we have plotted for each of the thirty different portfolio sizes the best evaluation obtained *in all 100 repetitions* (and not as in for the convergence experiment the average over the 100 best from each repetition). The meaning behind this is of course that an optimisation should lead to one recommendation — the index tracking portfolio to be constructed. So this simulates how good the tracking error was in total. As a comparison, we have given the OLS estimated portfolio of 19 shares as well.

The conclusion to be drawn from figure 7.3 is, as expected, that the tracking error decreases with increasing portfolio size. This finding is perfectly in line with intuition and with financial theory. Still, we minimize the tracking error TE in period 1, what we really want is as low a tracking error RTE as possible in the following periods, especially in period 2. We also depict the confidence intervals at a level $\alpha = 5\%$ for the tracking error TE P1 that we constructed by calculating the average and the standard error of the 1000 portfolios for each of the thirty portfolio sizes. Last, we can observe that the OLS estimation did not yield a tracking error TE P1 as good as the one obtained via the genetic algorithm.

Recall that the scenario to imagine is that we possess of historical data to optimise on in order to obtain good tracking for the time to come and of which we do not have any information. One would try to construct at a certain date the best index tracking portfolio by using the information that has at one's disposal, for example historic stock prices. This tracking portfolio is then held a certain time and exposed to tracking errors. We simulate this situation

Figure 7.3: Tracking error TE in period P1 of the best individual in a replication as a function of the portfolio size

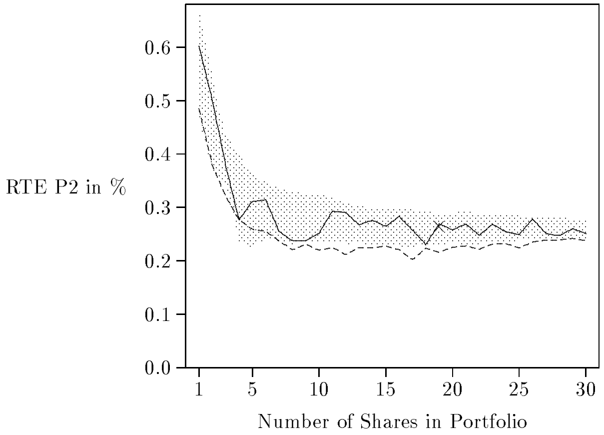


Source: Own calculation with the programmes `analyse` and `testpf`.
Note: The solid line shows the tracking error TE in period P1; the shaded region gives the $\alpha = 5\%$ confidence intervals for each portfolio size; the symbol $\times$ shows the tracking error TE P1 for the OLS estimated portfolio.

by dividing into several parts our stock price data. Period 1 (P1 for short) is used as the historic data that is known at the time the index tracking portfolio is constructed whereas periods two to four (P2, P3 and P4) are used as the unknown data at the time of the index fund construction. So to get an idea of the real performance of the index tracking portfolios, we have to consider the tracking error RTE in period P2 that is shown in figure 7.4.

There are several observations to make for the tracking error RTE in period P2 as depicted in figure 7.4. Let us begin with the solid line. It shows for each of the thirty portfolios that had the lowest tracking error TE in period P1 their respective tracking error RTE in period P2. This line is more erratic than the one for the TEs in PE1 that we showed in figure 7.3; the tracking errors increase several times when the portfolio is increased by another share — this is rather contrary to intuition and financial theory. But it only signals that this line gives the performance of individual portfolios, there is no sort of aggregation or filtering for the best ones. On the other hand, the shaded region shows the confidence intervals at a level of $\alpha = 5\%$ for the RTE P2. We obtained these by calculating the average and the standard error for the thousand RTE P2 that we possess for each portfolio size. (Thousand because we had 100 replication which each gave the ten best portfolios at the end of the run.) The development of this confidence region is rather regular. For increasing portfolio size, the confidence regions become smaller in size indicating that the expected tracking errors will be less dispersed for higher portfolio size. Furthermore, the confidence regions are also (after a certain portfolio size, say five) shifting downwards. It is surprising how little they are shifting downwards: one could expect only a marginally better tracking error for portfolio size of, say, 25 or 30 than for a rather small portfolio of size 10 or 15. So by increasing the size of the index tracking portfolio, you can only expect a tracking error with less dispersion but not

Figure 7.4: Tracking error RTE in period P2 of the best individual in a replication as a function of the portfolio size



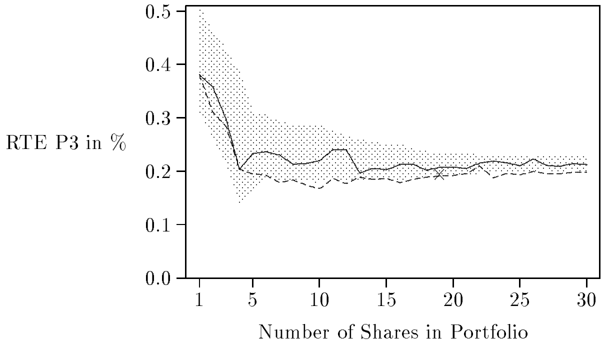
Source: Own calculation with the programmes `analyse` and `testpf`.
Note: The solid line shows the tracking error RTE in period P2 of the best portfolios for TE in P1; the shaded region gives the $\alpha = 5\%$ confidence intervals for each portfolio size; the dashed lines give as a comparison the best values obtained for RTE P2; the symbol $\times$ shows the tracking error RTE P2 for the OLS estimated portfolio.

necessarily a smaller tracking error. We also plotted the best observed tracking error RTE P2 as a dashed line for comparison. Their values are also quite interesting as they have a clear tendency to increase for increasing portfolio size; this is counterintuitive as well. Finally, it is certainly worth noting how well the OLS estimated portfolio did. Its tracking error value lies inside the 5% confidence interval for that portfolio size so that we cannot reject the null hypothesis that the genetic algorithm yields a tracking error (lower or) equal to that resulting from OLS estimation.

We next consider the tracking error RTE in period P3 that is shown in figure 7.5.

One would usually expect a slight degradation of all values because this period P3 is farther away from the initial in-sample period on which the optimisation was executed. So one is quite surprised to note the contrary — all observations are quite like the ones that we made apropos of figure 7.4. The solid line showing the tracking errors RTE P3 is erratic, the confidence intervals are shifting slowly downwards while decreasing their size, the reference values for the best RTE P3 show a minimum at a portfolio size $n = 10$. But there is one important difference to note: all values are smaller in magnitude. Whereas in figure 7.4, the RTEs were just under or at a value of 0.3%, their are now at or just above 0.2%. The OLS estimated portfolio nearly beats all values in the confidence intervals for $\alpha = 5\%$ with a value of under 0.2%. One also remarks the peak of the confidence interval at a portfolio size of four shares — there *can* be index tracking portfolios with a very low tracking error RTE in period P3 build from only four shares.

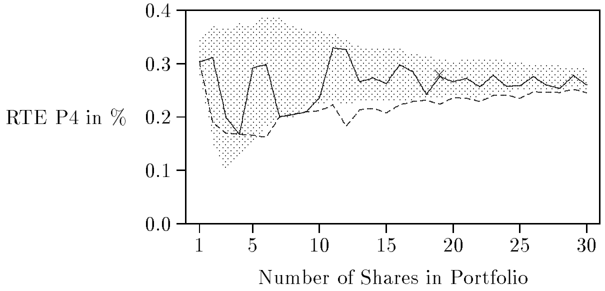
Figure 7.5: Tracking error RTE in period P3 of the best individual in a replication as a function of the portfolio size



Source: Own calculation with the programmes `analyse` and `testpf`.
Note: The solid line shows the tracking error RTE in period P3 of the best portfolios for TE in P1; the shaded region gives the $\alpha = 5\%$ confidence intervals for each portfolio size; the dashed lines give as a comparison the best values obtained for RTE P3; the symbol $\times$ shows the tracking error RTE P3 for the OLS estimated portfolio.

Last, we consider the tracking errors RTE in period P4.

Figure 7.6: Tracking error RTE in period P4 of the best individual in a replication as a function of the portfolio size



Source: Own calculation with the programmes `analyse` and `testpf`.
Note: The solid line shows the tracking error RTE in period P4 of the best portfolios for TE in P1; the shaded region gives the $\alpha = 5\%$ confidence intervals for each portfolio size; the dashed lines give as a comparison the best values obtained for RTE P4; the symbol $\times$ shows the tracking error RTE P4 for the OLS estimated portfolio.

Figure 7.6 shows that the comportment in the the third out-of-sample is worse than the two subsequent ones. The tracking errors depicted by the the solid line are even more dispersed, the confidence region is also wider as beforehand. The value for the OLS estimated portfolio is still very close to the respective one obtained by genetic algorithm index fund construction.

Which conclusions can be drawn for the use of genetic algorithms for index tracking fund construction? We first note that OLS estimation is a valid alternative. We obtained a portfolio of nineteen shares that tracks quite well the DAX that is constructed from a portfolio a thirty shares. There are two remarks to make. First, we can favourably compare the tracking errors obtained for this portfolio with the performance estimations from section 7.3. We there stated that every value between 0.2% and 0.5% would be a quite good one and any value lower than 0.2% would even be very satisfying taking into account the 0.2% tracking error for fully replicated funds. Well, OLS has beaten the 0.2% bound one time and scored clearly under 0.3% the two other times — so that is quite encouraging. But there remains a problem. Recall that we described the problem to be solved for index fund construction as a two-stage-process: you first search a subset of shares on which you can then run a quadratic minimization. If we reconsider figures 7.4 to 7.6, we see that portfolios of a size between seven and ,say, twelve performed very well. The index fund manager could thus track the DAX with a very small portfolio and would be exposed to tracking error of under 0.3% that he should be able to live with. OLS is different: we started with a full model of thirty variables where we extracted the shares that had either non-significative estimates or that had negative estimates. This yields a smaller model of which we can exclude more variables; this estimation – model reduction process can be repeated until the desired portfolio size equals the number of variables in the model. But OLS used this way "cannot go backwards", i.e. reinclude a share that was excluded beforehand. This has the advantage of reducing efficiently the size of the problem space — at the price of locking out portfolios with possible low tracking errors.

As to see whether an OLS model with a small number of variables results in a portfolio with a decent tracking error or not, we simply took our model from page 77. We then reduced in two steps all variables with an associated t-value that was smaller as 5.0. We thus obtained the model

$$\begin{aligned} DAX_t &= \underset{(10.18)}{0.1112}\,ALV_t + \underset{(11.73)}{0.1448}\,BAY_t + \underset{(6.36)}{0.0622}\,DAI_t + \underset{(7.57)}{0.1723}\,DBK_t \\ &+ \underset{(3.37)}{0.0509}\,DRB_t + \underset{(3.85)}{0.0177} +_t \underset{(5.17)}{0.1294}\,SIE_t + \underset{(4.72)}{0.0400}\,VIA_t \\ &+ \underset{(8.59)}{0.0446}\,VOW_t + \varepsilon_t\ . \end{aligned}$$

The sum of squared residuals equalled 1.6904, the explained sum of squares 369.9411; the null hypothesis of the non-significance of all parameters can still be rejected ($F = 1338.0$). So from the econometric viewpoint, this model is not as "good" as the preceding. We show its composition in table 7.8. The tracking errors for the nine-variable OLS portfolios are given in table 7.9. So the result is clear. Even though this model comprises less than half of the shares of the preceding one, the resulting tracking errors RTE P2 and RTE P3 are only slightly higher (0.28992 to 0.26386 and 0.20682 to 0.19360) as the corresponding ones from the nineteen-variables OLS model from page 77; the value for RTE P3 is even smaller (0.27563 to 0.27922). More important, compared to the genetic algorithm estimated portfolios, we see in reconsidering figures 7.4 to 7.6 that for RTE P3, OLS did better than the genetic algorithm whereas the latter won for RTE P1 and RTE P4. More generally, we see that the OLS values lie inside the confidence intervals, so once again we cannot reject the null hypothesis of equal

Table 7.8: Composition of the small portfolio estimated by OLS

```
# Portfolio from min.ol3
# from generation 3, evaluation 3 with value 0.007890
# Setup: 9 selectors, 9 weights, 9 effectively selected
ALV  0  14.026006   BAY  2  18.265874   DAI  8   7.844791   DBK 11  21.729533
DRB 12   6.415853   PRS 22   2.229489   SIE 25  16.320918   VIA 28   5.039057
VOW 29   8.128479
```

Source: Own computation with the programme `showpf` (cf. appendix B.6).
Note: The values in line two (generation number, evaluation number, evaluation value) are faked as this portfolio was not estimated by the GA; we simply needed some values to be in line with the data format required by `showpf`.

Table 7.9: Tracking Errors for the small OLS estimated portfolio

TE P1	RTE P2	RTE P3	RTE P4
0.20661	0.28992	0.20686	0.27563

Source: Own computations with the programme `analyse` (cf. appendix B.5).

values obtained by the two methods. So we cannot conclude that OLS does worse than the GA; we hence cannot claim to have *the* royal way to index fund construction. The genetic algorithm does find good tracking portfolios that did nearly as well on our DAX data nearly as did fully replicated funds. But OLS does not give tracking errors that are far worse, they might even be slightly better. So OLS *and* genetic algorithms seem to be valid methods for index fund construction.

And, of course, genetic algorithms might perform better than they did in our study. As this was not primarily an investigation into genetic algorithms, we neither tried to find new problem-specific operators or parameter values nor simply tried to use some of the operators that we described in chapter 3. We have presented genetic algorithms as a black box method for optimisation purposes — and we used them as a black box method. Another investigation into GA peak performance might show better results.

7.8 Summary

In this chapter, we

- revisited Chamberlain (1983) and Chamberlain and Rothschild (1983) in order to put the use of a variance-minimizing approach on safe ground; we also formulated our objective function in compact matrix notation and defined the tracking error that we use as objective function;
- we then considered the possible values that can — or cannot — be achieved when tracking an index; moreover, we cited some figures to have an idea of the tracking error for real-world index funds;
- we then reconsidered the method use to minimize the tracking error and discussed how we can represent portfolios for the use with the GENESIS package for genetic algorithm use;

- next we estimated an index tracking portfolio by using ordinary least squares; this approach gave us some figures to compare with the results to be obtained later on; we furthermore introduced the "real" tracking error RTE vis-à-vis the real market index;

- we then examined whether the genetic algorithm does converge for the problem under consideration; we compared the tracking errors for the two experiments with the ones from the OLS estimation and concluded prudently that the GA converges for the minimization of the tracking error in period P1;

- finally, we constructed index tracking portfolios for every possible portfolio size; with increasing portfolio size, the associated tracking error TE for period P1 were smaller, but unfortunately, this was not valid for the "real" tracking error RTE in periods P2 to P4; one there gets a smaller confidence region for a bigger portfolio size but only a slightly smaller RTE; we also saw that the results for the OLS estimated portfolio, though it does not give as low a TE P1, does give RTE values for P2 to P4 that are sufficiently small; we cannot reject the null hypothesis of same RTE values for the two methods.

Conclusion

You say you got a real solution
Well you know
we'd all love to change the plan
JOHN LENNON AND PAUL MCCARTNEY (1968)

With this work, we evaluated the possible use of genetic algorithms for constructing portfolios that track a given stock market index.

We introduced genetic algorithms in a simple example. We then rederived and presented some of the results that have been given in the literature on genetic algorithms; these could not entirely satisfy the demand for "water-tight" results that explain how genetic algorithms work. Next we discussed performance enhancements for genetic algorithms, i.e. different operators. So we can conclude that genetic algorithm provide an alternative for optimisation problems, especially when the problem domain is vast or/and not structured. But as a "blind" method, they cannot be expected to perform as well as adapted methods that make use of problem-specific knowledge.

Afterwards, we looked at some results from financial economics in order to elaborate some notions of financial analysis. We then discussed indexation or passive management as it is equally called and the methods for index fund construction that have been presented in the literature. We saw that indexation has its roots in the discussion on market efficiency — as there are no free lunches to gain in the market, fund managers cannot be expected to outperform the market average constantly and one hence is content with ensuring oneself the average market return by tracking the index. Thereafter, we considered the market index of the German stock exchange in Frankfurt/Main, the DAX and its formulae.

Finally, in the last chapter, we worked out the variance-minimizing approach and our tracking error definitions. We next discussed bounds for the tracking error that result principally from the fact that an index is simply a mathematical construct that is not exposed to some obstacles in real life as for example transaction costs. Thereafter, we estimated a tracking portfolio by using ordinary least squares (OLS), a technique dating back to the works of C.F. Gauß from the early nineteenth century. We then examined whether the genetic algorithm can be said to converge on two problems that each have a known solution, the index portfolio itself; we could prudently answer this question in the affirmative. Last, we tried to find the best portfolios for each possible portfolio size; we optimised on data from the first part of our data base and verified the best portfolios for this period on the remaining data. In the light of the tracking error bounds, the genetic algorithm does perform nicely; it constantly did better

than the index fund construction methods other than full replication. And surprisingly, the OLS estimated portfolio did as well as the genetic algorithms.

We can state to have found two valid methods for index fund construction not yet reported in the financial literature.

Appendix A

On GENESIS

N.B.: A version number is defined at the very end of this file;
please change that number whenever the file is modified!
And don't modify the file under any circumstances.
DONALD E. KNUTH (1987)

GENESIS was written by John Grefenstette to study genetic algorithms for function optimisation. The first development was carried out already in 1981. After the first presentation in 1984, it has it been widely distributed to the research community. This was made possible by Grefenstette's decision do allow the programme to be copied without charges provided that its use is limited to research or educational purposes. GENESIS has served as the basis for many of the genetic algorithm results since 1985. The version that we used for this work is version 5.0 as of February 1991 (see Grefenstette, Davis and Cerys 1991).

GENESIS is written is the C programming language defined by Kernighan and Ritchie (1988). This language is now widely recognized. Its small kernel makes it more machine independent than other programming languages — it is thus the preferred language for the development of programmes intended to run on a variety of computer types. GENESIS consists of twenty files of source code together with four header files of common definitions and data types.[1] Two makefiles for DOS and UNIX computer systems together with a set of six test functions complete the package.[2]

Table A.1 gives a brief description of each file.

The modularity of GENESIS allows to change the GA easily. If one wants to, say, experiment on another crossover operator, one writes another file `cross.c` using the same function interface and runs `make` to compile this file and to link it with the rest of GENESIS. The current standard of GENESIS for its genetic algorithm operations is two-point crossover, selection

[1] It is common practice in C not to create very large files compromising several functions but rather small files, preferably one for each major function. A header file usually contains directives for the language pre-processor that will be included in the source files. So instead of copying code around between source files, one writes a header file that is automatically included at compilation. The random number generator that we presented is a good example for code from a header file. Header files have the suffix `.h`.

[2] A makefile contains informations about the dependencies between the files for the make utility, a driver for the compiler and the linker that, following the dependencies, calls these two former ones after changes have been made to the source files.

Table A.1: The GENESIS genetic algorithm package

Filename	Description
`best.c`	input, maintenance and output of best structures
`checkpnt.c`	save global variables in a file for later restart
`convert.c`	functions that translate between various representations
`cross.c`	perform two-point crossover on entire population
`define.h`	global definitions (data types, random number generator)
`display.c`	manages the display for interactive use
`done.c`	tests experiment-termination conditions
`elitist.c`	lets the best element survive in the next generation
`error.c`	print message in error log file and abort
`eval.c`	problem specific user-defined objective function
`evaluate.c`	evaluate the current population by calling the user-defined function `eval`
`extern.h`	external declarations (i.e. parameters specified by the user)
`format.h`	formats the input and output data files
`generate.c`	form a new generate, evaluate and gather performance statistics
`global.h`	the global variables shared by several files
`init.c`	create an initial population and initialize performance variables
`input.c`	read the input files and initialize variables
`main.c`	main programme for GENESIS
`measure.c`	calculate performance measures
`mutate.c`	perform mutation on the current population
`report.c`	calculate mean and variance of some of performance measures
`restart.c`	restart an interrupted GA run
`schema.c`	measure the current allocation of trials to a schema
`select.c`	choose a new population (selection and sampling)
`setup.c`	create an input file for GENESIS

Source: The GENESIS programme was written by Grefenstette (see Grefenstette, Davis and Cerys 1991).

either rank based or roulette wheel type and Baker's sampling algorithm along with several handy in- and output features independent of the actual GA operations.

There are three executable files resulting from these source files: `ga`, `select` and `report`. As `ga` obviously starts a genetic algorithm run, we have to say some words about the other two programmes. It is quite common in the UNIX world to store the parameters of a programme in plain ASCII-text files that can be read and edited with any other programme. GENESIS comes with the same idea; there could be up to eleven files associated for the in- and output of a GA experiment. In order to distinguish the files belonging to different experiments, they can be grouped with a common file suffix. We will suppose the suffix `.mem` for this section. The programme `select` prompts the user for each parameter value for the input files `in.mem` and `template.mem`. The former contains all the parameter values and options determining a GA run: number of experiments, number of evaluations per experiment, population size, crossover and mutation rate, several values for scaling and replacing options as the generation gap as well as options that determine how many "best guys" are saved during the run and at the end of the experiment. Furthermore, among the options, there are choices for the kind of performance statistics and generation dumps, for possible use of the elitist replacement strategy, for a floating point representation, for Gray coding representation, for rank base selection, for schema tracing and for the possible initialization of a generation via given values in an `init.mem` file. As an example, we show in table A.2 the `in`-file for the example used in chapter 1.

If a floating point representation is chosen, the `select` programme demands the number of genes (i.e. number of elements in the vector passed to user-specified evaluation function), their

Table A.2: The `in.mem`-file used in chapter 1

```
        Experiments = 1
       Total Trials = 1000
    Population Size = 10
   Structure Length = 8
     Crossover Rate = 0.6
      Mutation Rate = 0.001
     Generation Gap = 1.0
     Scaling Window = 5
    Report Interval = 100
   Structures Saved = 10
 Max Gens w/o Eval = 2
      Dump Interval = 0
        Dumps Saved = 0
            Options = cefIMR
        Random Seed = 123456789
           Rank Min = 0.75
```

Source: Own GENESIS application.

minimal and maximal values and their the number of values between (i.e. what determines then implicitly the number of bits used to represent these values with the binary alphabet). These coding informations are stored in the `template.mem` file. This is a rather handsome option because the GENESIS user does not have to bother about binary representation; all this is done by the programme. We show the `template.mem` from chapter 1 in table A.3.

Table A.3: The `template.mem`-file used in chapter 1

```
genes: 1

gene 0
min: 0
max: 1
values: 256
format: %8.5lf
```

Source: Own GENESIS application

The other executable programme, `report`, provides an ex-post analysis after a GA run. Because of the dependence of a genetic algorithm on the random number generator, one usually repeats an experiment several times and analyzes the output generated this way by letting `report` calculate all the means and variances of the performance measures. These are

- generation number,
- trial number (which is not simply 'generation' × 'population size' as GENESIS recognizes when an offspring is identical to a population member so that no evaluation is necessary),
- the figure 'conv' which is defined as the number of positions in the bitstring in which at least 95% of the population has the same value,
- the figure 'lost' which is defined as the number of positions in the bitstring in which 100% of the population has the same value,

- the figure 'bias' that indicated the average percentage of the most prominent value in each position; i.e. a bias of 75% means that on average, each position has converged to 75% of `0` or 75% of `1`,
- on-line performance as defined by De Jong (1975) (see page 26) as the average of all individuals created so far,
- off-line performance as defined by De Jong (1975) (see page 26) as the average of all generation-best individuals,
- the evaluation of the generation-best individual and finally
- the average evaluation of the population.

It is therefore necessary to set one of the two options that enable convergence measurements; then these measurements are dumped to the file `out` at an user-specified interval. `report` then calculates mean and variance of several different performance values as on-line, off-line, best and average performance.

GENESIS comes with six simple evaluation functions. As an example, we give in table A.4 the source code of the first test function provided by GENESIS. In order to maintain the

Table A.4: GENESIS' first example for an evaluation function

```
/************************************************  file f1.c  ****/
double eval(str, length, vect, genes)
char str[];     /* string representation                        */
int length;     /* length of bit string                         */
double vect[];  /* floating point representation                */
int genes;      /* number of elements in vect                   */
{
      register int i;
      double sum;

      sum = 0.0;
      for (i = 0; i < genes; i++)
         sum += vect[i] * vect[i];

      return (sum);
}
/************************************************ end of file ****/
```

Source: File `f1.c` from GENESIS (Grefenstette, Davis and Cerys 1991)

black box character, the user does not change the definition of the interface to GENESIS. Thus, among the four function parameters, there are always two redundant ones because one either works without binary representation (i.e. with the bit string `str` stored as an array of characters of size `length`) or with floating point representation (i.e. the array `vect` of double-precision floating point values of size `genes`). This evaluation function used floating point representation for a simple parabola of dimension `genes`.

With such an easy example (that, of course, every hill-climbing algorithm would solve easily), the user can toggle all the different parameter values and options to get a "feeling" for the behaviour of a GA. The GENESIS system is described in greater detail in the users guide in Grefenstette, Davis and Cerys (1991) as well as the object-orientated genetic algorithm OOGA written in LISP by Davis and Cerys.

Appendix B

The Source Files

Well, my terminal's locked up, and I ain't got any mail,
And I can't recall the last time that my program didn't fail,
I've got stacks in my structs, I've got arrays in my queues,
I've got the Segmentation Violation Core Dumped Blues.
found in the Computer Science Department at Karlsruhe, May 1992

We presented in the last section the GENESIS package that makes is quite easy to use a genetic algorithm for a user-specified problem. We will in this section present all the source files that we have written for the index fund construction application of genetic algorithms.

B.1 `master` — Running the GA Experiments

This programme `master` reads in one textfile that has the name `master.abc` in which the user can specify the values of the options influencing the genetic algorithms behaviour for this GA run that is grouped under the suffix `abc`. `master` then writes the files `in.abc` and `template.abc` and starts the actual genetic algorithm programme. After the termination of the genetic algorithm, `master` evokes, if one of the options '`c`' or '`C`' that command output of the convergence statistics in the `out.abc` file, the programme `report` that sums up these convergence statistics; the datafile `out.abc` that forms the input for `report` is afterwards deleted. Finally `analyse` is evoked for the calculation of the tracking errors of the portfolios dumped at the end of each repetition.

B.1.1 `master.h`

This file contains the declarations of the file in- and output formats used with the files `in` and `master`.

```
/***********************************************************************
*                                                                      *
*   Program     GAPM - a Genetic Algorithm for Passive Management      *
*                                                                      *
*   File        master.h                                               *
*                                                                      *
*   Function    preprocessor statements, types etc for master.c        *
```



```
*               and analyse.c                                          *
*                                                                      *
*   Parameter   none                                                   *
*                                                                      *
*   Caller      none (but included in master.c and analyse.c)          *
*                                                                      *
*   Uses        nothing else                                           *
*                                                                      *
*   Author      Dirk Eddelbuettel                                      *
*                                                                      *
*   Date        Jan 1992 and Jul 1992                                  *
*                                                                      *
***********************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

                                        /* Preprocessor declarations    */
                                        /* from 'global.h'              */

                                        /* input file is read in accor- */
                                        /* ding to IN_FORMAT and IN_VARS*/
#define MASTER_FORMAT " \
   Version Number = %d \
 Number Selectors = %d \
 Selector Minimum = %f \
 Selector Maximum = %f \
  Selector Values = %u \
Selector PrintStr = %s \
   Number Weights = %d \
   Weight Minimum = %f \
   Weight Maximum = %f \
    Weight Values = %u \
  Weight PrintStr = %s \
 Normalize Switch = %d \
      Experiments = %d \
     Total Trials = %d \
  Population Size = %d \
 Structure Length = %d \
   Crossover Rate = %lf \
    Mutation Rate = %lf \
   Generation Gap = %lf \
   Scaling Window = %d \
  Report Interval = %d \
 Structures Saved = %d \
Max Gens w/o Eval = %d \
    Dump Interval = %d \
      Dumps Saved = %d \
          Options = %s \
      Random Seed = %lu \
         Rank Min = %lf "

#define MASTER_VARS &iVersion, &iNumSelectors,\
        &fMinSelectors, &fMaxSelectors,\
        &uValSelectors, sValPrtStr,\
        &iNumWeights, &fMinWeights, &fMaxWeights,\
        &uValWeights, sWeiPrtStr, &iNormalizeSwitch, \
        &Totalexperiments,&Totaltrials,&Popsize,&Length,\
        &C_rate,&M_rate,&Gapsize,\
        &Windowsize,&Interval,\
        &Savesize,&Maxspin,\
        &Dump_freq,&Num_dumps,\
        Options,\
        &OrigSeed, &Rank_min
```

```
                                        /* OUT_FORMAT is the format for */
                                        /* printing the input parameters*/
                                        /* for GENESIS/GAPM             */
#define OUT_FORMAT "\
      Experiments = %d\n\
     Total Trials = %d\n\
  Population Size = %d\n\
 Structure Length = %d\n\
   Crossover Rate = %7.5f\n\
    Mutation Rate = %7.5f\n\
   Generation Gap = %7.5f\n\
   Scaling Window = %d\n\
  Report Interval = %d\n\
 Structures Saved = %d\n\
Max Gens w/o Eval = %d\n\
    Dump Interval = %d\n\
      Dumps Saved = %d\n\
          Options = %s\n\
      Random Seed = %lu\n\
         Rank Min = %7.5f\n"

                                        /* OUT_VARS are the parameters  */
                                        /* to be printed according to   */
                                        /* OUT_FORMAT                   */
#define OUT_VARS Totalexperiments,Totaltrials,Popsize,Length,\
        C_rate,M_rate,Gapsize,\
        Windowsize,Interval,Savesize,Maxspin,\
        Dump_freq,Num_dumps,Options,OrigSeed,Rank_min

                                        /* These variables definitions  */
                                        /* are copied from 'global.h'   */
                                        /* The Input file specifies     */
                                        /* these parameters             */

   int    Totalexperiments;             /* number of experiments        */
   int    Totaltrials;                  /* trials per experiment        */
   int    Popsize;                      /* population size              */
   int    Length;                       /* bit length of a structure    */
   double C_rate;                       /* crossover rate               */
   double M_rate;                       /* mutation rate                */
   double Gapsize;                      /* fract. pop replaced per gen  */
   int    Windowsize;                   /* used to update worst perf.   */
   int    Interval;                     /* trials between statistics    */
   int    Savesize;                     /* num. of struct. in minfile   */
   int    Maxspin;                      /* max gens without evals       */
   int    Dump_freq;                    /* gens between checkpointing   */
   int    Num_dumps;                    /* num. of checkpoint files kept*/
   char   Options[40];                  /* option flags                 */
   unsigned long    OrigSeed;           /* original value for seed      */
   double Rank_min;                     /* min. sampling rate (if 'r')  */


   int iVersion;                        /* to check corresponding file  */
   int iNumSelectors,
       iNumWeights;
   float fMinSelectors,
       fMinWeights,
       fMaxSelectors,
       fMaxWeights;
   unsigned uValSelectors,
       uValWeights;
   char sValPrtStr[20],
       sWeiPrtStr[20];
   int iNormalizeSwitch;
```


B.1.2 master.c

```
/**********************************************************************
*                                                                     *
*   Program    GAPM - a Genetic Algorithm for Passive Management      *
*                                                                     *
*   File       master.c                                               *
*                                                                     *
*   Function   'master' forms the shell around the GAPM/GENESIS appli- *
*              cation. It writes the parameter files 'template' and   *
*              and 'in' and serves thus as a test platform for        *
*              multiple calls of GAPM.                                *
*                                                                     *
*   Parameter  argc, *argv[]  for suffix                              *
*                                                                     *
*   Caller     none - gets called form the system shell by the user   *
*                                                                     *
*   Uses       nothing apart from standard C functions                *
*                                                                     *
*   Author     Dirk Eddelbuettel                                      *
*                                                                     *
*   Date       Jan 92 and Jul 92                                      *
*                                                                     *
**********************************************************************/

#include "define.h"                     /* defines UNIX or TURBOC       */
#include "master.h"                     /* file input/output declarat.  */
#include "eval.h"                       /* file i/o to eval()           */

void main(int argc, char *argv[])
{
   FILE *fp;
   char cParameterFile[20],
        cInFileName[20],
        cTemplateFileName[20],
        cOutFileName[20],
        cReportFileName[20],
        cAnalyseFileName[20],
        cDelMinFiles[20],
        cEvalParFile[] = "eval.par";
   char cSuffix[4];
   char cCommand[80];
   register int i;
   double dCon;


   if (argc != 2)                       /* Is a suffix specified ?      */
      {
      fprintf(stderr,"Master for GAPM-calls\n");
      fprintf(stderr,"Syntax:  master <xxx> \t\t");
      fprintf(stderr,"with xxx as file suffix for the experiments\n");
      abort();
      }
                                        /* Build filenames with suffix  */
   sprintf(cSuffix, "%3.3s", argv[1]);
   sprintf(cParameterFile, "master.%3.3s", cSuffix);
   sprintf(cInFileName, "in.%3.3s", cSuffix);
   sprintf(cTemplateFileName, "template.%3.3s", cSuffix);
   sprintf(cReportFileName, "report.%3.3s", cSuffix);
   sprintf(cOutFileName, "out.%3.3s", cSuffix);
   sprintf(cAnalyseFileName, "analyse.%3.3s", cSuffix);
   #if TURBOC
       sprintf(cDelMinFiles, "del min*.%3.3s >nul", cSuffix);
   #else
       sprintf(cDelMinFiles, "rm min*.%3.3s", cSuffix);
   #endif
```

```
                                     /* Read own parameter file      */
if ((fp = fopen(cParameterFile, "r")) == NULL)
   {
   fprintf(stderr, "Master: can't open %s for reading\n", cParameterFile);
   abort();
   }
fscanf(fp,MASTER_FORMAT,MASTER_VARS);/* see master.h for types+vars  */
fclose(fp);

                                     /* basic consitency tests       */
if (iVersion != 2)
   {
   fprintf(stderr, "Master: Parameter file is not matching\n");
   abort();
   }

                                     /* Check, if Length is correct  */
if (iNumSelectors != 0)
   dCon = log((double)uValSelectors)/log(2.0) * iNumSelectors;
else
   dCon = 0;
if (iNumWeights != 0)
   dCon += log((double)uValWeights)/log(2.0) * iNumWeights;

if (Length != (int) dCon)
   {
   fprintf(stderr,"Master: Parameters not consistent (length)\n");
   abort();
   }

                                     /* everything is allright, so   */
                                     /* start by deleting old stuff  */
remove(cInFileName);
remove(cTemplateFileName);
remove(cReportFileName);
remove(cOutFileName);
remove(cAnalyseFileName);
remove(cEvalParFile);
system(cDelMinFiles);                /* can't remove(min-*.xyz) with */
                                     /* wildcard; hence directly del.*/

                                     /* Write parameter file for eval*/
                                     /* -> simply copy master.<abc>  */
if ((fp = fopen(cEvalParFile, "w")) == NULL)
   {
   fprintf(stderr, "Master: can't open %s for writing\n", cEvalParFile);
   abort();
   }
fprintf(fp,EVAL_FORMAT,EVAL_OUT_VARS);   /* see eval.h for definition*/
fclose(fp);


                                     /* Construction of 'template.x' */
if ((fp = fopen(cTemplateFileName, "w")) == NULL)
   {
   fprintf(stderr, "Master: can't open %s for writing\n",
                    cTemplateFileName);
   abort();
   }
fprintf(fp,"genes: %d\n", iNumSelectors + iNumWeights);

for (i=0; i<iNumSelectors; i++)      /* Write all selectors genes    */
   {
   fprintf(fp, "\ngene %d\n", i);
   fprintf(fp, "min: %f\n", fMinSelectors);
   fprintf(fp, "max: %f\n", fMaxSelectors);
```



```
   fprintf(fp, "values: %u\n", uValSelectors);
   fprintf(fp, "format: %s\n", sValPrtStr);
   }

                                     /* and all weight genes         */
for (i=iNumSelectors; i<iNumSelectors+iNumWeights; i++)
   {
   fprintf(fp, "\ngene %d\n", i);
   fprintf(fp, "min: %f\n", fMinWeights);
   fprintf(fp, "max: %f\n", fMaxWeights);
   fprintf(fp, "values: %u\n", uValWeights);
   fprintf(fp, "format: %s\n", sWeiPrtStr);
   }
fclose(fp);                          /* now we've got a template     */



                                     /* Construction of 'in.x'       */
if ((fp = fopen(cInFileName, "w")) == NULL)
   {
   fprintf(stderr, "Master: can't open %s for writing\n", cInFileName);
   abort();
   }
fprintf(fp,OUT_FORMAT,OUT_VARS);
fclose(fp);


                                     /* now call GAPM with suffix    */
sprintf(cCommand, "gapm %3.3s", cSuffix);
system(cCommand);
                                     /* if option `c' or `C' chosen, */
                                     /* let's make a report          */
if (strpbrk("cC", Options))
   {
   sprintf(cCommand, "report %3.3s > report.%3.3s", cSuffix, cSuffix);
   system(cCommand);
   remove(cOutFileName);             /* saves up to 4 MByte !!       */
   }
                                     /* and an analyse               */
sprintf(cCommand, "analyse %3.3s > analyse.%3.3s", cSuffix, cSuffix);
system(cCommand);

sprintf(cCommand, "mkdir run%3.3s", cSuffix);
system(cCommand);
                                     /* clean up a little            */
#if TURBOC
    sprintf(cCommand, "copy *.%3.3s run%3.3s", cSuffix, cSuffix);
    system(cCommand);
    sprintf(cCommand, "del *.%3.3s >nul", cSuffix);
    system(cCommand);
#else
    sprintf(cCommand, "mv *.%3.3s run%3.3s", cSuffix, cSuffix);
    system(cCommand);
#endif
}
```

B.2 `eval` — Evaluate a GA Individual

The function `eval()` forms part of the genetic algorithm programme. When using GENESIS, all you have to do is to write a such a function that receives from the genetic algorithm an individual from the current population, evaluates is and returns the fitness value to the GA. Here, we thus calculate the tracking error TE for the structure handed over from the GA.

eval does initialize itself at the first call. It then allocates all the memory, read the data files, sets up the return covariance matrix and calculates some constants. For every subsequent call, all that has to be done is then to multiply the vector of shares with the covariance matrix and with another vector.

B.2.1 eval.h

This file contains some definitions used by the function eval().

```
/************************************************************************
*                                                                       *
*   Program    GAPM - a Genetic Algorithm for Passive Management        *
*                                                                       *
*   File       eval.h                                                   *
*                                                                       *
*   Function   Deklaration of preproc. constants and types for eval()   *
*                                                                       *
*   Parameter  none                                                     *
*                                                                       *
*   Caller     none (but this file is included in some other files...) *
*                                                                       *
*   Uses       no other functions                                       *
*                                                                       *
*   Author     Dirk Eddelbuettel                                        *
*                                                                       *
*   Date       Nov/Dec 1991                                             *
*                                                                       *
************************************************************************/


                                        /* includes these header files  */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>


                                        /* preprocessor constants       */

#define PRICE_FILENAME  "stockpri.dax"  /* Name of file with price data */
#define NUMBER_FILENAME "stocknmb.dax"  /* Name of file with factor data*/
#define MAX_STOCKS      30              /* max. 30 stocks               */
#define SERIES_LENGTH   100             /* length of a cal. time series */

                                        /* we cannot pass anything to  */
                                        /* eval() which is not in line */
                                        /* with the function def, hence*/
                                        /* parameterfile               */
#define EVAL_FORMAT " \
   Version Number = %d \
 Number Selectors = %d \
   Number Weights = %d \
 Normalize Switch = %d "

#define EVAL_OUT_VARS iVersion, \
        iNumSelectors,\
        iNumWeights, \
        iNormalizeSwitch

#define EVAL_IN_VARS &iVersion, \
        &iNumSelectors,\
        &iNumWeights, \
        &iNormalizeSwitch
```


B.2.2 eval.c

The file contains the functions eval() and vNormalize().

```
/************************************************************************
*                                                                       *
*   Program    GAPM - a Genetic Algorithm for Passive Management        *
*                                                                       *
*   File       eval.c                                                   *
*                                                                       *
*   Function   evaluates one structure (one genetic algorithm           *
*              individual) handed over by GENESIS;                      *
*              eval() reads (only once of course) the share data files *
*              and then sets up all data structures                     *
*              the objective is to minimize the variance of the         *
*              difference between the DAX-return and the return of     *
*              a synthetic portfolio that is determined by the          *
*              structure given from GENESIS; eval() actually            *
*              calculates this variance of the difference and returns  *
*              its square root, the standard error, that is defined    *
*              as the tracking error                                    *
*                                                                       *
*   Parameter  char str[]     bit representation of one pop. member    *
*              int len        length of str[]                           *
*              double vect[]  vector with float. repres. of str[]      *
*              int genes      number of elements in vect[]              *
*                                                                       *
*   Caller     GENESIS from the file evaluate.c                         *
*                                                                       *
*   Uses       vNormalize()   in this file                              *
*              vRead_Data()   in dax_data.c                             *
*              vGetPhi()      in dax_data.c                             *
*              vGetMemForData()in dax_data.c                            *
*              vCalcCovMat    in dax_data.c                             *
*              vMatProd()     in matrix.c                               *
*              vMatScMlt()    in matrix.c                               *
*              vScMlt()       in matrix.c                               *
*                                                                       *
*   Author     Dirk Eddelbuettel                                        *
*                                                                       *
*   Date       V1: Nov/Dec 1991                                         *
*              V2: Feb/Mar 1992                                         *
*              V3: Jly/Aug 1992                                         *
*                                                                       *
************************************************************************/
#include "eval.h"                       /* include header file          */
#include "dax_data.h"                   /* share data type              */
#include "matrix.h"                     /* matrix types and macros      */

                                        /* Tests for several perform.   */
                                        /* limits/lower bounds          */
                                        /* Exactly one on the following */
                                        /* ifdef's must one 1           */
                                        /* to select exactly one version*/
                                        /* of 'vNormalize()'            */

#define LIM_NORMAL  1                   /* normal mode                  */
#define LIM_INIT_1  0                   /* initialize with day 1 values */
                                        /* all selectors + weights      */
#define LIM_INIT_SL 0                   /* initialize with day SER._LEN.*/
                                        /* values all sel's + weights   */

double eval(char str[], int len, double vect[], int genes)
{
   static int iFirstCall = 1;           /* Flag for first call of func. */
   static AllData Data;                 /* structure for all the data   */
   static Matrix *mC, *mP;              /* Matrix and vector for var    */
```

```
Matrix *mPhi, *mVec;                  /* tmp.vecs for const.var.part  */
double *dPhi;                         /* temp. vector to read Phis    */
static int *iShare;                   /* vector for index of share    */
static double *dPercent;              /* vector for percent. of share */
static double dConstVarPart = 0.0;    /* const. part in var expression*/
double dVar;                          /* variance                     */
FILE   *fp;                           /* file pointer for data file   */
char   cParamFile[] = "eval.par";     /* = master.<abc>               */
static int iVersion;                  /* to check corresponding file  */
static int iNumSelectors;             /* # selectors (from paramfile) */
static int iNumWeights;               /* # weights (from paramfile)   */
static int iNormalizeSwitch;          /* convergence test??           */
register int i,j;                     /* loop variables               */


                                      /* Prototypes                   */
void vNormalize(double[], int[], double [], int, int, int);
void vGetPhi(AllData *, double *, int);
void vMatProd(Matrix*, Matrix*, Matrix*);
void vMatScMlt(Matrix*, double, Matrix*);
void vScProd(Matrix *, Matrix *, double *);
int iGetMemForData(AllData *);
int iReadData(AllData *, const char *, const char *);
void vCalcCovMat(AllData *, int, int, Matrix *);
double sqrt(double);

                                      /* is this the first time ?     */
if (iFirstCall)                       /* if yes, we've got some work  */
   {
   iFirstCall = 0;                    /* there's only one first time  */

   Data.iNSh = 30;                    /* 30 shares in the DAX         */
   Data.iObs = 247;                   /* 247 observations per comp    */

                                      /* Read own parameter file      */
   if ((fp = fopen(cParamFile, "r")) == NULL)
      {
      fprintf(stderr, "eval(): can't open %s for reading\n", cParamFile);
      abort();
      }
   fscanf(fp,EVAL_FORMAT,EVAL_IN_VARS);  /* see eval.h for def.s     */
   fclose(fp);

                                      /* check consistency            */
   if (  (iVersion != 2)
       &&(genes != iNumSelectors + iNumWeights))
      {
      fprintf(stderr, "eval(): problem with parameters\n");
      abort();
      }

   if (iNormalizeSwitch == 1)         /* is it the convergence test ? */
      iNumSelectors = 30;             /* value was zero for GENESIS   */
                                      /* but we need 30 here          */

                                      /* get all the memory           */
   MAKE_MATRIX(mC,Data.iNSh,Data.iNSh);      /* covariance matrix    */
   MAKE_MATRIX(mP,Data.iNSh,1);       /* var = xCx + px , this is p   */
   MAKE_MATRIX(mPhi,Data.iNSh,1);     /* to determine p, we need phi  */
   MAKE_MATRIX(mVec,Data.iNSh,1);     /* for const. var part          */
   iShare   = (int*)    calloc((size_t) iNumSelectors, sizeof(int));
   dPercent = (double*) calloc((size_t) iNumWeights, sizeof(double));
   dPhi     = (double*) calloc((size_t) Data.iNSh, sizeof(double));
   if (  (iGetMemForData(&Data))      /* if memory problem in func.   */
       || !mC || !mP || !mPhi || !mVec
       || !iShare || !dPercent || !dPhi )
```



```
      {
      fprintf(stderr,"ERROR, but not enough memory for eval !!");
      abort();
      }

                                      /* read data files              */
   if (iReadData(&Data, PRICE_FILENAME, NUMBER_FILENAME))
      {
      fprintf(stderr,"Data incorrectly read !!");
      abort();
      }

                                      /* calc C, phi, p und phiCphi   */
   vCalcCovMat(&Data, 1, SERIES_LENGTH, mC);
   vGetPhi(&Data,dPhi,SERIES_LENGTH);/* get phis from function        */
   for (i=0; i<Data.iNSh; i++)        /* copy phis into vector for    */
      VEC(mPhi,i) = dPhi[i];          /* all shares in index          */
   free(dPhi);
   vMatProd(mC, mPhi, mP);            /* 'p = C * phi'                */
   vMatScMlt(mP, -2.0, mP);           /* p = p * (-2)                 */
   vMatProd(mC,mPhi,mVec);            /* calc phi C phi               */
   vScProd(mVec,mPhi,&dConstVarPart);
   FREE_MATRIX(mPhi);
   FREE_MATRIX(mVec);
   }                                  /* end for the 1st time part    */


                                      /* now what is done everytime   */

vNormalize(vect, iShare, dPercent,
                    iNormalizeSwitch, iNumSelectors, iNumWeights);


                                      /* var = xCx + phiCphi + px     */
dVar = dConstVarPart;                 /* set var to 'phi C phi'       */
for (i=0; i<iNumWeights; i++)         /* for all shares considered    */
   {                                  /* ie for all i,j elem. of P    */
                                      /* var = xCx + px               */
   for (j=0; j<iNumWeights; j++)
      dVar +=   dPercent[i] * dPercent[j] * MAT(mC, iShare[i], iShare[j]);
   dVar += dPercent[i] * VEC(mP,iShare[i]);
   }

dVar = sqrt(dVar);                    /* get the standard error       */
return(dVar);                         /* return `the tracking error'  */
}                                     /* end for eval()               */

#if LIM_NORMAL
void vNormalize(double vec[], int iSh[], double dPr[],
                          int iNormSw, int iNSel, int iNWei)
{
register int i;                       /* loop variable                */
double dSum = 0.0;                    /* sum of relative share values */

if (iNormSw == 0)                     /* normal mode ?                */
   {
   for (i=0; i<iNSel; i++)            /* for all "share selectors"    */
      if ((int) vec[i] >= MAX_STOCKS)/* if GENESIS choose a value     */
         {                            /* that does not have a corresp.*/
         vec[i+iNSel] = 0.0;          /* stock in our list, we set the*/
                                      /* volume of this stock to 0.0  */
         iSh[i] = 0;                  /* and take another one instead */
         }
      else                            /* else values has a share      */
         iSh[i] = (int) vec[i];       /* copy from "vec" to "iSh"     */
```

```
      for (i=iNSel; i<iNSel+iNWei; i++) /* for all "rel. share values"  */
         dSum += vec[i];                /* sum up                       */

      if (dSum == 0.0)                  /* we had only illegal values   */
         dSum = 1.0;                    /* this won't do no harm as the */
                                        /* indiv. pos. are zero !       */

      for (i=0; i<iNSel; i++)           /* calc. percentage values      */
         dPr[i] = vec[i+iNSel]/dSum;    /* Percent. = value / sum   */
      }


   else if (iNormSw == 1)               /* convergence experiments ?    */
      {
      for (i=0; i<iNWei; i++)           /* for all "share selectors"    */
         iSh[i] = i;                    /* impose the share i           */

      for (i=0; i<iNWei; i++)           /* for all "rel. share values"  */
         dSum += vec[i];                /* sum up                       */

      if (dSum == 0.0)                  /* we had only illegal values   */
         dSum = 1.0;                    /* this won't do no harm as the */
                                        /* indiv. pos. are zero !       */

      for (i=0; i<iNWei; i++)           /* calc. percentage values      */
         dPr[i] = vec[i] / dSum;        /* Percent. = value / sum       */
      }
   else                                 /* problem !!                   */
      {
      fprintf(stderr,"\n\n vNormalize(): problem with normalize switch!\n");
      abort();
      }
}
#endif                                  /* #if LIM_NORMAL               */


                                        /* ATTENTION---this is the      */
                                        /* for the DAX-convergence test */
                                        /* All values handed in by      */
                                        /* GENESIS are overwritten by   */
                                        /* constant values of 03/09/90  */

                                        /* There are diff.versions for  */
                                        /* diff.tests - see the #def's  */

#if LIM_INIT_1
                                        /* Values as calculated by      */
                                        /* vGetPhi( , , 1); this gives  */
                                        /* lower bound for the track.err*/
void vNormalize(double vec[], int iSh[], double dPr[], int iNSel, int iNWei)
{
   register int i;
   double sum = 0.0;

   if (gen != 60)
      {
      printf("\n\n vNormalize(): value of gen = %d and not 60!!\n",gen);
      abort();
      }

   iSh[ 0] =  0; dPr[ 0] = 0.1247;      /* Allianz      */
   iSh[ 1] =  1; dPr[ 1] = 0.0395;      /* BASF         */
   iSh[ 2] =  2; dPr[ 2] = 0.0465;      /* Bayer        */
   iSh[ 3] =  3; dPr[ 3] = 0.0175;      /* Bay.HypoBank */
   iSh[ 4] =  4; dPr[ 4] = 0.0225;      /* BMW          */
```



```
   iSh[ 5] =  5; dPr[ 5] = 0.0139;      /* Bay.Ver.Bank */
   iSh[ 6] =  6; dPr[ 6] = 0.0197;      /* Commerzbank  */
   iSh[ 7] =  7; dPr[ 7] = 0.0079;      /* Continental  */
   iSh[ 8] =  8; dPr[ 8] = 0.0957;      /* DaimlerBenz  */
   iSh[ 9] =  9; dPr[ 9] = 0.0075;      /* Degussa      */
   iSh[10] = 10; dPr[10] = 0.0031;      /* Dt.Babcock   */
   iSh[11] = 11; dPr[11] = 0.0896;      /* DeutscheBank */
   iSh[12] = 12; dPr[12] = 0.0442;      /* DresdnerBank */
   iSh[13] = 13; dPr[13] = 0.0106;      /* Henkel       */
   iSh[14] = 14; dPr[14] = 0.0408;      /* Hoechst      */
   iSh[15] = 15; dPr[15] = 0.0146;      /* Karstadt     */
   iSh[16] = 16; dPr[16] = 0.0166;      /* Kaufhof      */
   iSh[17] = 17; dPr[17] = 0.0151;      /* Linde        */
   iSh[18] = 18; dPr[18] = 0.0116;      /* Lufthansa    */
   iSh[19] = 19; dPr[19] = 0.0163;      /* MAN          */
   iSh[20] = 20; dPr[20] = 0.0234;      /* Mannesmann   */
   iSh[21] = 21; dPr[21] = 0.0129;      /* Metallgesell.*/
   iSh[22] = 22; dPr[22] = 0.0145;      /* Preussag     */
   iSh[23] = 23; dPr[23] = 0.0545;      /* RWE          */
   iSh[24] = 24; dPr[24] = 0.0131;      /* Schering     */
   iSh[25] = 25; dPr[25] = 0.0933;      /* Siemens      */
   iSh[26] = 26; dPr[26] = 0.0215;      /* Thyssen      */
   iSh[27] = 27; dPr[27] = 0.0464;      /* VEBA         */
   iSh[28] = 28; dPr[28] = 0.0165;      /* VIAG         */
   iSh[29] = 29; dPr[29] = 0.0461;      /* Volkswagen   */

   for (i=0; i<30; i++)
      sum += dPr[i];

   if (sum < 0.999 || sum > 1.001)
      {
      printf("\n\n vNormalize(): sum of percentages = %lf"
             " and not 100.0!\n",sum);
      abort();
      }
}
#endif                                  /* #if LIM_INIT_1               */


#if LIM_INIT_SL
                                        /* Values as calculated by      */
                                        /* vGetPhi( , , SERIES_LENGTH); */
                                        /* this gives another lower     */
                                        /* lower bound for the track.err*/
void vNormalize(double vec[], int iSh[], double dPr[], int iNSel, int iNWei)
{
   register int i;
   double sum = 0.0;

   if (gen != 60)
      {
      printf("\n\n vNormalize(): value of gen = %d and not 60!!\n",gen);
      abort();
      }

   iSh[ 0] =  0; dPr[ 0] = 0.1275;      /* Allianz      */
   iSh[ 1] =  1; dPr[ 1] = 0.0410;      /* BASF         */
   iSh[ 2] =  2; dPr[ 2] = 0.0506;      /* Bayer        */
   iSh[ 3] =  3; dPr[ 3] = 0.0175;      /* Bay.HypoBank */
   iSh[ 4] =  4; dPr[ 4] = 0.0211;      /* BMW          */
   iSh[ 5] =  5; dPr[ 5] = 0.0140;      /* Bay.Ver.Bank */
   iSh[ 6] =  6; dPr[ 6] = 0.0208;      /* Commerzbank  */
   iSh[ 7] =  7; dPr[ 7] = 0.0070;      /* Continental  */
   iSh[ 8] =  8; dPr[ 8] = 0.0867;      /* DaimlerBenz  */
   iSh[ 9] =  9; dPr[ 9] = 0.0076;      /* Degussa      */
   iSh[10] = 10; dPr[10] = 0.0036;      /* Dt.Babcock   */
```

```
   iSh[11] = 11; dPr[11] = 0.0907;      /* DeutscheBank */
   iSh[12] = 12; dPr[12] = 0.0419;      /* DresdnerBank */
   iSh[13] = 13; dPr[13] = 0.0093;      /* Henkel       */
   iSh[14] = 14; dPr[14] = 0.0430;      /* Hoechst      */
   iSh[15] = 15; dPr[15] = 0.0139;      /* Karstadt     */
   iSh[16] = 16; dPr[16] = 0.0139;      /* Kaufhof      */
   iSh[17] = 17; dPr[17] = 0.0155;      /* Linde        */
   iSh[18] = 18; dPr[18] = 0.0108;      /* Lufthansa    */
   iSh[19] = 19; dPr[19] = 0.0182;      /* MAN          */
   iSh[20] = 20; dPr[20] = 0.0268;      /* Mannesmann   */
   iSh[21] = 21; dPr[21] = 0.0106;      /* Metallgesell.*/
   iSh[22] = 22; dPr[22] = 0.0115;      /* Preussag     */
   iSh[23] = 23; dPr[23] = 0.0562;      /* RWE          */
   iSh[24] = 24; dPr[24] = 0.0162;      /* Schering     */
   iSh[25] = 25; dPr[25] = 0.1024;      /* Siemens      */
   iSh[26] = 26; dPr[26] = 0.0199;      /* Thyssen      */
   iSh[27] = 27; dPr[27] = 0.0471;      /* VEBA         */
   iSh[28] = 28; dPr[28] = 0.0178;      /* VIAG         */
   iSh[29] = 29; dPr[29] = 0.0369;      /* Volkswagen   */

   for (i=0; i<30; i++)
      sum += dPr[i];

   if (sum < 0.999 || sum > 1.001)
      {
      printf("\n\n vNormalize(): sum of percentages = %lf"
             " and not 100.0!\n",sum);
      abort();
      }
}
#endif                                  /* #if LIM_INIT_SL             */
```

B.3 daxdata — Data Handling Routines

`daxdata` contains several functions used by `eval()`, for example the actual memory allocation for the stock market data structure, the actual data reading and the calculation of the covariance matrix.

B.3.1 daxdata.h

This file contains the data type used for the stock market data.

```
/***********************************************************************
*                                                                      *
*   Program     GAPM - a Genetic Algorithm for Passive Management      *
*                                                                      *
*   File        daxdata.h                                              *
*                                                                      *
*   Function    Deklaration of prepr. const. and types for daxdata()   *
*                                                                      *
*   Parameter   none                                                   *
*                                                                      *
*   Caller      none (but this file is included in some other files...) *
*                                                                      *
*   Uses        no other functions                                     *
*                                                                      *
*   Author      Dirk Eddelbuettel                                      *
*                                                                      *
*   Date        Jan and Jul 92                                         *
*                                                                      *
***********************************************************************/
```



```
#define  SHARE  float                   /* might be changed to double   */

typedef struct                          /* data struct for one share    */
   {
   char sName[10];                      /* share has a name             */
   SHARE *fP;                           /* a vector full of prices      */
   SHARE *fN;                           /* and a vector with 'numbers'  */
   }
OneShare;

typedef struct                          /* struct for 'all the data'    */
   {
   int iNSh;                            /* number of shares in index    */
   int iObs;                            /* number of prices read        */
   OneShare *Sh;                        /* vector of share structs      */
   char sIndex[10];                     /* name of the index            */
   SHARE *fInd;                         /* vector for index values      */
   char **sDate;                        /* vector for date strings      */
   }
AllData;
```

B.3.2 daxdata.c

```
/***********************************************************************
*                                                                      *
*   Program     GAPM - a Genetic Algorithm for Passive Management      *
*                                                                      *
*   File        daxdata.c                                              *
*                                                                      *
*   Function    'daxdata' contains the functions for the data file     *
*               handling for the dax data:                             *
*               iReadData() that reads the data files                  *
*               vCheckDax() that verifies the data base                *
*               vGetPhi()   that calculates the phi-factors            *
*               iGetMemForData() that allocates the memory             *
*               vFreeData() that disposes the allocated memory         *
*               dStdErrDiff() that calculates the std for 2 time series *
*               vRMSEDiff() that calculates the RMSE for 2 time series *
*               vCalcCovMat() that calculates the covariance matrix    *
*                                                                      *
*   Caller      eval() and analyse()                                   *
*                                                                      *
*   Uses        nothing apart from standard C functions                *
*                                                                      *
*   Author      Dirk Eddelbuettel                                      *
*                                                                      *
*   Date        Jan 92 and Jul 92                                      *
*                                                                      *
***********************************************************************/

#include <stdio.h>
#include <stdlib.h>
#if TURBOC
#include <alloc.h>
#endif
#include "dax_data.h"
#include "matrix.h"



/* iReadData  : read share data file
   input      : pointer to structure AllData for all the data
                filename for stock price data
```

```
               filename for 'number of shares' for each share, each day
   output     : structure with the data read
   return     : 0 if no error occurred
               1 if an (fscanf or file) error occurred
   uses       : fopen(), fscanf()
   note       : THIS IS A HACK! BE SURE TO GIVE TO RIGHT FILES ALONG WITH
               THE CORRECT PARAMETERS !                               */
int iReadData(AllData *Dt, const char *sPrF, const char *sNmF)
{
   int iOK = 1;                         /* Success = 1; Failure != 1    */
   FILE *file;                          /* filepointer for reading      */
   register int ii;                     /* loop variable                */
   SHARE fHelp;                         /* -> bug in TurboC++ !!        */

   if ((file = fopen(sPrF, "r")) == NULL)
      {
      fprintf(stderr,"ERROR: Can't open file  %s\n",sPrF);
      return(1);
      }

   iOK = (4 == fscanf(file,"%*s %s %s %s %s", &Dt->sIndex, &Dt->Sh[0].sName,
                            &Dt->Sh[11].sName, &Dt->Sh[12].sName));
   for (ii=0; iOK && ii<Dt->iObs; ii++)
      {
      iOK = (5 == fscanf(file, "%s %*f %f %f %f %f", Dt->sDate[ii],
                            &fHelp, &(Dt->Sh[0].fP[ii]),
                            &(Dt->Sh[11].fP[ii]), &(Dt->Sh[12].fP[ii])));
      Dt->fInd[ii] = fHelp;
      }

   iOK = (6 == fscanf(file, "%s %s %s %s %s %s",
                            &Dt->Sh[6].sName, &Dt->Sh[5].sName,
                            &Dt->Sh[3].sName, &Dt->Sh[20].sName,
                            &Dt->Sh[26].sName, &Dt->Sh[10].sName));
   for (ii=0; iOK && ii<Dt->iObs; ii++)
      iOK = (6 == fscanf(file, "%f %f %f %f %f %f",
                            &(Dt->Sh[6].fP[ii]),  &(Dt->Sh[5].fP[ii]),
                            &(Dt->Sh[3].fP[ii]),  &(Dt->Sh[20].fP[ii]),
                            &(Dt->Sh[26].fP[ii]), &(Dt->Sh[10].fP[ii])));

   iOK = (6 == fscanf(file, "%s %s %s %s %s %s",
                            &Dt->Sh[19].sName, &Dt->Sh[17].sName,
                            &Dt->Sh[25].sName, &Dt->Sh[1].sName,
                            &Dt->Sh[2].sName, &Dt->Sh[14].sName));
   for (ii=0; iOK && ii<Dt->iObs; ii++)
      iOK = (6 == fscanf(file, "%f %f %f %f %f %f",
                            &(Dt->Sh[19].fP[ii]),  &(Dt->Sh[17].fP[ii]),
                            &(Dt->Sh[25].fP[ii]),  &(Dt->Sh[1].fP[ii]),
                            &(Dt->Sh[2].fP[ii]), &(Dt->Sh[14].fP[ii])));

   iOK = (6 == fscanf(file, "%s %s %s %s %s %s",
                            &Dt->Sh[24].sName, &Dt->Sh[9].sName,
                            &Dt->Sh[13].sName, &Dt->Sh[23].sName,
                            &Dt->Sh[27].sName, &Dt->Sh[28].sName));
   for (ii=0; iOK && ii<Dt->iObs; ii++)
      iOK = (6 == fscanf(file, "%f %f %f %f %f %f",
                            &(Dt->Sh[24].fP[ii]),  &(Dt->Sh[9].fP[ii]),
                            &(Dt->Sh[13].fP[ii]),  &(Dt->Sh[23].fP[ii]),
                            &(Dt->Sh[27].fP[ii]), &(Dt->Sh[28].fP[ii])));
   iOK = (6 == fscanf(file, "%s %s %s %s %s %s",
                            &Dt->Sh[15].sName, &Dt->Sh[16].sName,
                            &Dt->Sh[4].sName, &Dt->Sh[8].sName,
                            &Dt->Sh[29].sName, &Dt->Sh[7].sName));
   for (ii=0; iOK && ii<Dt->iObs; ii++)
      iOK = (6 == fscanf(file, "%f %f %f %f %f %f",
                            &(Dt->Sh[15].fP[ii]),  &(Dt->Sh[16].fP[ii]),
```



```
                            &(Dt->Sh[4].fP[ii]),  &(Dt->Sh[8].fP[ii]),
                            &(Dt->Sh[29].fP[ii]), &(Dt->Sh[7].fP[ii])));

   iOK = (3 == fscanf(file, "%s %s %s",
                            &Dt->Sh[21].sName, &Dt->Sh[22].sName,
                            &Dt->Sh[18].sName));
   for (ii=0; iOK && ii<Dt->iObs; ii++)
      iOK = (3 == fscanf(file, "%f %f %f",
                            &(Dt->Sh[21].fP[ii]),  &(Dt->Sh[22].fP[ii]),
                            &(Dt->Sh[18].fP[ii])));
   fclose(file);

   if (iOK)
      if ((file = fopen(sNmF, "r")) == NULL)
         {
         fprintf(stderr,"ERROR: Can't open file  %s\n",sNmF);
         return(1);
         }
   iOK = (0 == fscanf(file,"%*s %*s %*s %*s %*s %*s"));
   for (ii=0; iOK && ii<Dt->iObs; ii++)
      iOK = (5 == fscanf(file, "%*s %f %f %f %f %f", &(Dt->Sh[0].fN[ii]),
                            &(Dt->Sh[11].fN[ii]),&(Dt->Sh[12].fN[ii]),
                            &(Dt->Sh[6].fN[ii]), &(Dt->Sh[5].fN[ii])));

   iOK = (0 == fscanf(file,"%*s %*s %*s %*s %*s %*s"));
   for (ii=0; iOK && ii<Dt->iObs; ii++)
      iOK = (6 == fscanf(file, "%f %f %f %f %f %f",
                            &(Dt->Sh[3].fN[ii]), &(Dt->Sh[20].fN[ii]),
                            &(Dt->Sh[26].fN[ii]),&(Dt->Sh[10].fN[ii]),
                            &(Dt->Sh[19].fN[ii]), &(Dt->Sh[17].fN[ii])));

   iOK = (0 == fscanf(file,"%*s %*s %*s %*s %*s %*s"));
   for (ii=0; iOK && ii<Dt->iObs; ii++)
      iOK = (6 == fscanf(file, "%f %f %f %f %f %f",
                            &(Dt->Sh[25].fN[ii]), &(Dt->Sh[1].fN[ii]),
                            &(Dt->Sh[2].fN[ii]),&(Dt->Sh[14].fN[ii]),
                            &(Dt->Sh[24].fN[ii]), &(Dt->Sh[9].fN[ii])));

   iOK = (0 == fscanf(file,"%*s %*s %*s %*s %*s %*s"));
   for (ii=0; iOK && ii<Dt->iObs; ii++)
      iOK = (6 == fscanf(file, "%f %f %f %f %f %f",
                            &(Dt->Sh[13].fN[ii]), &(Dt->Sh[23].fN[ii]),
                            &(Dt->Sh[27].fN[ii]), &(Dt->Sh[28].fN[ii]),
                            &(Dt->Sh[15].fN[ii]), &(Dt->Sh[16].fN[ii])));

   iOK = (0 == fscanf(file,"%*s %*s %*s %*s %*s %*s"));
   for (ii=0; iOK && ii<Dt->iObs; ii++)
      iOK = (6 == fscanf(file, "%f %f %f %f %f %f",
                            &(Dt->Sh[4].fN[ii]), &(Dt->Sh[8].fN[ii]),
                            &(Dt->Sh[29].fN[ii]), &(Dt->Sh[7].fN[ii]),
                            &(Dt->Sh[21].fN[ii]), &(Dt->Sh[22].fN[ii])));

   iOK = (0 == fscanf(file,"%*s"));
   for (ii=0; iOK && ii<Dt->iObs; ii++)
      iOK = (1 == fscanf(file, "%f",&(Dt->Sh[18].fN[ii])));

   fclose(file);

   return ( ! iOK);                     /* 0: no Error, =! 0: Error      */
}


/* vCheckDAX  : compare DAX read from file to DAX calculated with formula
   input      : pointer to structure AllData for all the data
   output     :  -nothing-  (result is printed on screen)
   return     :  -nothing-
```

```
   uses       : fabs()
   note       : this routine was written to verify the data base; there is
                a slight mismatch for some days that is probably due to
                the manually created factor file (factor F, see eq.(6.2)*/
void vCheckDAX(AllData *Dt)
{
   register int ii, ij;
   double dPF, dErr;
   double fabs(double);

   puts("\n\n\"Uberpr\"ufung der gelesenen Daten");
   puts("Dax \t Pf \t err \t date");
   for (ii=0; ii<Dt->iObs; ii++)
      {
      dPF = 0.0;
      for (ij=0; ij<Dt->iNSh; ij++)
         dPF += Dt->Sh[ij].fP[ii] * Dt->Sh[ij].fN[ii];
      dPF /= 29356.73 * 0.001;
      dErr = ((double) Dt->fInd[ii] - dPF) / dPF * 100.0;
      if (fabs(dErr) > 0.05)
         printf("%7.2f  &  %7.2f  & %8.5f  & %s\n", Dt->fInd[ii], dPF,
                               dErr, Dt->sDate[ii]);
      }
}

/* vGetPhi    : determines the share percentages in the index
   input      : pointer to structure AllData for all pointers to data
                day for which the percentage are to be determined
   output     : vector with percentage values
   return     :  -nothing-
   uses       :  -nothing-
   note       : we haven't got p and p*q from 1987 better than that; the
                formula is eq. (6.3) in section 6.4                    */
void vGetPhi(AllData *Dt, double *dP, int iDay)
{
   register int i;

   for (i=0; i<Dt->iNSh; i++)
      dP[i] = Dt->Sh[i].fN[iDay]        /* factor 'F' from DAX formula  */
                 * 34287.8              /* sum of capital in 1987       */
                 / 10065790.53          /* sum of (p*q) in 1987         */
                 * Dt->Sh[i].fP[iDay]   /* price                        */
                 / Dt->fInd[iDay]       /* DAX value                    */
                 * 1000.0               /* base value 1000 on 31/12/87  */
                 / 100.0;               /* normalize. 5% => 0.05        */
}

/* iGetMemForData : Allocation of memory for all market data
   input      : pointer to structure AllData for all pointers to data
                number of shares for which data is going to be read
                number of observations (or simply days with trade)
   output     : structure with the correct amount of mem at each pointer
   return     : 0 if no error occurred
                1 if an (memory allocation error occurred)
   uses       : calloc()
   note       : this not very elegant (two calls for every share, one call
                for every date string) but we had problems with MS-DOS'
                'no more than 64kB in one allocation' restriction      */
int iGetMemForData(AllData *Data)
{
   int iErr = 0;                        /* Errorcode                    */
   register int ii;                     /* loop variable                */

                                        /* build vector of structs for  */
                                        /* vectors to share data        */
   iErr +=  (Data->Sh = (OneShare *)calloc((size_t) Data->iNSh,
```



```
                                        sizeof(OneShare))) == NULL;

                                        /* for every share, get memory  */
                                        /* for price and number vectors */
   for (ii=0; !iErr && ii<Data->iNSh; ii++)
       {
       iErr += (Data->Sh[ii].fP = (SHARE *)calloc((size_t) Data->iObs,
                                        sizeof(SHARE))) == NULL;
       iErr += (Data->Sh[ii].fN = (SHARE *)calloc((size_t) Data->iObs,
                                        sizeof(SHARE))) == NULL;
       }


   if (!iErr)                           /* vector for index data        */
      iErr += (Data->fInd = (SHARE *)calloc((size_t) Data->iObs,
                                        sizeof(SHARE))) == NULL;

   if (!iErr)                           /* vector of dates for each day */
      iErr += (Data->sDate = (char **)calloc((size_t) Data->iObs,
                                        sizeof(char*))) == NULL;

                                        /* for each day, get string mem */
   for (ii=0; !iErr && ii<Data->iObs; ii++)
      iErr += (Data->sDate[ii] = (char*)calloc((size_t) 10,
                                        sizeof(char))) == NULL;

   return iErr;
}

/* vFreeData  : set memory for struct AllData free
   input      : pointer to structure AllData for all pointers to data
   output     :  -nothing-
   return     :  -nothing-
   uses       : free()
   note       : ---                                                     */
void vFreeData(AllData *Data)
{
   register int i;                      /* loop variable                */



   for (i=0; i<Data->iNSh; i++)         /* for every share, free memory */
      {
      free(Data->Sh[i].fP);
      free(Data->Sh[i].fN);
      }
   free(Data->Sh);                      /* free share vector            */
   free(Data->fInd);                    /* free index vector            */
   for (i=0; i<Data->iObs; i++)         /* for each day, free date      */
      free(Data->sDate[i]);
   free(Data->sDate);                   /* free date vector             */
}


/* dStdErrDiff: determine standard error of difference of the two series;
                std.err is calculated using the normal equation
   input      : vector with series of values (ie returns)
                vector with series of values (ie returns)
                start index in return series (starting at that value)
                end index in return series (ending at that value)
   output     :  -nothing-
   return     : standard error calculated
   uses       : double()
   note       :  ---                                                    */
double dStdErrDiff(double *dSer1, double *dSer2, int iSt, int iEnd)
{
```

```
   register int i;
   double dVar = 0.0;
   double dMean = 0.0;
   double sqrt(double);

   for (i=iSt; i<=iEnd; i++)            /* calculate E(IR - PR)         */
      dMean += (dSer1[i] - dSer2[i]);
   dMean /= (iEnd - iSt + 1);

   for (i=iSt; i<=iEnd; i++)            /* calc E((IR-PR) - E(IR_PR))^2 */
      dVar += (dSer1[i]-dSer2[i]-dMean)*(dSer1[i]-dSer2[i]-dMean);
   dVar /= (iEnd - iSt + 1);

   return(sqrt(dVar));
}

/* dRMSEDiff  : determine root mean square error of the difference
                of the two series;
   input      : vector with series of values (ie returns)
                vector with series of values (ie returns)
                start index in return series (starting at that value)
                end index in return series (ending at that value)
   output     :  -nothing-
   return     : root mean square error
   uses       : double()
   note       :  ---                                                    */
double dRMSEDiff(double *dSer1, double *dSer2, int iSt, int iEnd)
{
   register int i;
   double dMSE = 0.0;
   double sqrt(double);

   for (i=iSt; i<=iEnd; i++)
      dMSE += (dSer1[i]-dSer2[i])*(dSer1[i]-dSer2[i]);
   dMSE /= (iEnd - iSt + 1);

   return(sqrt(dMSE));
}

/* vCalcCovMat: calculate the covariance matrix for the optimization period
   input      : pointer to structure AllData for all the data
                start of calculation period (inclusively)
                end of calculation period (inclusively)
   output     : covariance matrix of type Matrix (cf. matrix.h)
   return     :  -nothing-
   uses       :  -nothing-
   note       : Memory for output matrix must be allocated by caller !!
                ATTENTION: we will calc a return for day iSt, ie refer
                to iST-1 so do NOT call with iST=0                      */
void vCalcCovMat(AllData *Dt,int iSt,int iEn, Matrix *mC)
{
   register int i,j,k;
   int iLen = iEn - iSt + 1;
   double dCov;
   Matrix *mRetMn, *mRet;

   MAKE_MATRIX(mRetMn,Dt->iNSh,1);      /* return mean for each share   */
   MAKE_MATRIX(mRet,Dt->iNSh,iLen);     /* return each share, each day  */
   if ( !mRetMn || !mRet )
      {
      puts("ERROR, but not enough memory for cov. mat. !!");
      exit(-1);
      }
                                        /* build return matrix  and     */
                                        /* avg. return vector           */
   for (i=0; i<Dt->iNSh; i++)           /* for all shares               */
```



```
      {
      VEC(mRetMn,i) = 0;
      for (j=0; j<iLen; j++)            /* for all dates                */
         {
         MAT(mRet,i,j) = (double) 100.0 *
                         (Dt->Sh[i].fP[j+iSt+1] - Dt->Sh[i].fP[j+iSt])
                         / (Dt->Sh[i].fP[j+iSt]);
         VEC(mRetMn,i) += MAT(mRet,i,j);/* return mean for stock j      */
         }
      VEC(mRetMn,i) /= iLen;            /* average return of share i    */
      }

                                        /* now build covariance matrix  */
                                        /* 'C' with the returns         */
   for (i=0; i<Dt->iNSh; i++)           /* for all shares               */
      {
      for (j=i; j<Dt->iNSh; j++)        /* for all resting shares       */
         {
         dCov = 0.0;
         for (k=0; k<iLen; k++)         /* for all days                 */
            dCov +=   (MAT(mRet,i,k) - VEC(mRetMn,i))
                    * (MAT(mRet,j,k) - VEC(mRetMn,j));

         dCov /= iLen;                  /* accord. to TE-definition     */
         MAT(mC,i,j) = dCov;            /* go thru vect. as thru matrix */
         MAT(mC,j,i) = dCov;            /* over positions that corresp. */
                                        /* their virtual matr. positions*/
         }
      }
   FREE_MATRIX(mRet);                   /* don't need returns any more  */
   FREE_MATRIX(mRetMn);                 /* nor their means              */
}
```

B.4 matrix — General Matrix Calculations

This file contains some general purpose matrix calculation routines.

B.4.1 matrix.h

Here we define the data type of a matrix.

```
/************************************************************************
*                                                                       *
*   Program     GAPM - a Genetic Algorithm for Passive Management       *
*                                                                       *
*   File        matrix.h                                                *
*                                                                       *
*   Function    Deklaration of prepr. constants and types for matrix()  *
*                                                                       *
*   Parameter   none                                                    *
*                                                                       *
*   Caller      none (but this file is included in some other files...) *
*                                                                       *
*   Uses        no other functions                                      *
*                                                                       *
*   Author      Dirk Eddelbuettel                                       *
*                                                                       *
*   Date        Feb 92                                                  *
*                                                                       *
************************************************************************/

#define ETYPE   double                  /* for one matrix element       */
```

```
                                        /* this is the most precise     */
                                        /* possible !                   */


typedef struct                          /* struct for a matrix          */
{
   ETYPE *EV;                           /* vector of elements           */
   int iNR;                             /* number of rows               */
   int iNC;                             /* number of colons             */
} Matrix;

                                        /* Macro for matrix adressing   */
                                        /* this will pick this element  */
                                        /* in row 'i' and colon 'j'     */
                                        /* representation of matrix is  */
                                        /* a vector of length = row*col */
#define MAT(m,i,j)      (m->EV[ (int) (i) * m->iNC + j ])
                                        /* and for vect.of type 'Matrix'*/
#define VEC(m,i)        (m->EV[ (int) (i) ])

                                        /* Get memory for the matrix    */
                                        /* pointer p and thereafter     */
                                        /* for the matrix of dim r*c    */
                                        /* that p points at             */
#define MAKE_MATRIX(p,r,c)  {p = (Matrix*)calloc(1, sizeof(Matrix));\
                             p->iNR = r;\
                             p->iNC = c;\
                             p->EV  = (ETYPE*)calloc((size_t) ((r)*(c)),\
                                        sizeof(ETYPE));}

#define FREE_MATRIX(p)      {free(p->EV);\
                             free(p);}
```

B.4.2 matrix.c

```
/**************************************************************************
*                                                                         *
*   Program     GAPM - a Genetic Algorithm for Passive Management         *
*                                                                         *
*   File        matrix.c                                                  *
*                                                                         *
*   Function    'matrix' contains some matrix calculation functions:      *
*               vMatInvert() inverts a matrix                             *
*               iGetMatMem() allocates memory for a matrix                *
*               vMatTransp() transposes a matrix                          *
*               vMatProd()   mutliplies two matrices                      *
*               vMatAdd()    adds a matrix to another                     *
*               vMatSub()    substracts a matrix from another             *
*               vMatScMlt()  multiplies a matrix by a scalar              *
*               vMatScAdd()  adds a scalar to eacj cell of a matrix       *
*               vScProd()    multiplies two vectors                       *
*                                                                         *
*   Caller      eval() and analyse()                                      *
*                                                                         *
*   Uses        nothing apart from standard C functions                   *
*                                                                         *
*   Author      Dirk Eddelbuettel                                         *
*                                                                         *
*   Date        Feb 92                                                    *
*                                                                         *
**************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
```



```
#if TURBOC
#include <alloc.h>
#include <sys\timeb.h>                  /* struct timeb() hier dekl.    */
#endif

#include "matrix.h"                     /* matrix types and macros      */

#define TRUE    1
#define FALSE   0

/* vMatInvert : Inversion of a quadratic matrix
   input      : pointer to matrix THAT MUST BE QUADRATIC!
   output     : determinant of the input-matrix as a pointer to double
                pointer to inverted matrix
   return     :  - nothing -
   uses       : no functions but two macros in order to use the vectors
                as a (quadr.) matrix representations; advantage is the
                the unlimited size (!!) of the matrixes to be handled
                (this should be proper & portable C code)
   note       : NO ERROR CHECKING IS DONE IN THE FUNCTION
                IF YOU HANDING IN A NON-QUADRATIC MATRIX......          */
void vMatInvert(Matrix *Mmat, double *dDet, Matrix *Minv)
{
   int *iPivRow,                        /* vector of rows for pivoting  */
       *iPivCol,                        /* vector of cols for pivoting  */
       *iPivCheck;                      /* vector for pivot TRUE/FALSE  */
   register int i,j,k;                  /* loop variables               */
   int iRow,iCol;                       /* for one cell position        */
   int iDim = Mmat->iNR;                /* dimension of matrix          */
   ETYPE dPiv, dMult, dLeval, dSwap;    /* misc.                        */

   Minv->iNC = Mmat->iNC;
   Minv->iNR = Mmat->iNR;
   iPivRow   = (int *)calloc((size_t) Mmat->iNR, sizeof(int));
   iPivCol   = (int *)calloc((size_t) Mmat->iNR, sizeof(int));
   iPivCheck = (int *)calloc((size_t) Mmat->iNR, sizeof(int));

   *dDet = 1.0;
   for (i=0; i<iDim; i++)
      {
      iPivCheck[i] = FALSE;     /* well, with calloc, that isn't necess. */
      for (j=0; j<iDim; j++)
         MAT(Minv,i,j) = MAT(Mmat,i,j);
      }

   for (i=0; i<iDim; i++)
      {
      dLeval = 0.0;
      for (j=0; j<iDim; j++)
         if ( ! iPivCheck[j])
            for (k=0; k<iDim; k++)
               if ( ! iPivCheck[k])
                  if ( fabs(MAT(Minv,j,k)) > dLeval )
                     {
                     iRow = j;
                     iCol = k;
                     dLeval = fabs(MAT(Minv,j,k));
                     }

      iPivCheck[iCol] = TRUE;
      iPivRow[i] = iRow;
      iPivCol[i] = iCol;

      if (iRow != iCol)
         {
         *dDet *= -1.0;
```

```
         for (j=0; j<iDim; j++)
            {
            dSwap = MAT(Minv,iRow,j);
            MAT(Minv,iRow,j) = MAT(Minv,iCol,j);
            MAT(Minv,iCol,j) = dSwap;
            }
         }

      dPiv = MAT(Minv,iCol,iCol);
      *dDet *= dPiv;
      if (*dDet > 1.0e30)               /* against numeric overflow     */
         *dDet = 1.0;
      MAT(Minv,iCol,iCol) = 1.0;
      for (j=0; j<iDim; j++)
         MAT(Minv,iCol,j) /= dPiv;

      for (j=0; j<iDim; j++)
         if (j != iCol)
            {
            dMult = MAT(Minv,j,iCol);
            MAT(Minv,j,iCol) = 0;
            for (k=0; k<iDim; k++)
               MAT(Minv,j,k) -= MAT(Minv,iCol,k) * dMult;
            }
      }                                 /* for (i=0; ....               */

   for (i=0; i<iDim; i++)
      {
      j = iDim - i - 1;
      if (iPivRow[j] != iPivCol[j])
         for (k=0; k<iDim; k++)
            {
            dSwap = MAT(Minv,k,iPivRow[j]);
            MAT(Minv,k,iPivRow[j]) = MAT(Minv,k,iPivCol[j]);
            MAT(Minv,k,iPivCol[j]) = dSwap;
            }
      }
}

/* iGetMatMem : Allocate memory for one matrix and store dimensions
   input      : number of rows
                number of colons
   output     : pointer to new matrix
   return     :  0 if everything was allright
                 1 if a (memory) error occured
   uses       : no other functions (apart from calloc())
   note       :  ---                                                    */
int iGetMatMem(int iRows, int iCols, Matrix *Mnew)
{
    Mnew->iNR   = iRows;
    Mnew->iNC   = iCols;
    Mnew->EV = (ETYPE*)calloc((size_t) iRows*iCols,
                                          sizeof(ETYPE));
    return (Mnew->EV == NULL);
}

/* vMatTransp : Transposes a matrix
   input      : pointer to matrix to be transposed
   output     : pointer to transposed matrix
   return     :  - nothing -
   uses       : no functions but a macro in order to use the vectors
                as a (quadr.) matrix representations; advantage is the
                the unlimited size (!!) of the matrixes to be handled
                (this should be proper & portable C code)
   note       : NO ERROR CHECKING IS DONE IN THE FUNCTION, MAKE SURE
                THAT IN THE MATRIX STRUCT STANDS THE CORRECT DIM.       */
```



```
void vMatTransp(Matrix *Mmat, Matrix *Mtrans)
{
   register int i,j;

   Mtrans->iNR = Mmat->iNC;
   Mtrans->iNC = Mmat->iNR;
   for (i=0; i<Mmat->iNR; i++)
      for (j=0; j<Mmat->iNC; j++)
         MAT(Mtrans,j,i) = MAT(Mmat,i,j);
}

/* vMatProd   : product of two matrices
   input      : pointer to input matrix 1
                pointer to input matrix 2
   output     : pointer to product matrix
   return     :  - nothing -
   uses       : no functions but a macro in order to use the vectors
                as a (quadr.) matrix representations; advantage is the
                the unlimited size (!!) of the matrixes to be handled
                (this should be proper & portable C code)
   note       : NO ERROR CHECKING IS DONE IN THE FUNCTION, MAKE SURE
                THAT IN THE MATRIX STRUCT STANDS THE CORRECT DIM.       */
void vMatProd(Matrix *Mmat1, Matrix *Mmat2, Matrix *Mprod)
{
   register int i,j,k;

   Mprod->iNR = Mmat1->iNR;
   Mprod->iNC = Mmat2->iNC;
   for (i=0; i<Mmat1->iNR; i++)
      for (j=0; j<Mmat2->iNC; j++)
         {
         MAT(Mprod, i, j) = 0.0;
         for (k=0; k<Mmat1->iNC; k++)
            MAT(Mprod,i,j) += MAT(Mmat1,i,k) * MAT(Mmat2,k,j);
         }
}

/* vMatAdd    : add two matrices
   input      : pointer to input matrix 1
                pointer to input matrix 2
   output     : pointer to result matrix
   return     :  - nothing -
   uses       : no functions but a macro in order to use the vectors
                as a (quadr.) matrix representations; advantage is the
                the unlimited size (!!) of the matrixes to be handled
                (this should be proper & portable C code)
   note       : NO ERROR CHECKING IS DONE IN THE FUNCTION, MAKE SURE
                THAT IN THE MATRIX STRUCT STANDS THE CORRECT DIM.       */
void vMatAdd(Matrix *Mmat1, Matrix *Mmat2, Matrix *Madd)
{
   register int i,j;

   Madd->iNR = Mmat1->iNR;
   Madd->iNC = Mmat1->iNC;
   for (i=0; i<Mmat1->iNR; i++)
      for (j=0; j<Mmat1->iNC; j++)
         MAT(Madd, i, j) = MAT(Mmat1,i,j) + MAT(Mmat2,i,j);
}

/* vMatSub    : subtract one matrix from another
   input      : pointer to input matrix 1
                pointer to input matrix 2
   output     : pointer to result matrix = matrix1 - matrix2
   return     :  - nothing -
   uses       : no functions but a macro in order to use the vectors
                as a (quadr.) matrix representations; advantage is the
```

```
                the unlimited size (!!) of the matrixes to be handled
                (this should be proper & portable C code)
   note       : NO ERROR CHECKING IS DONE IN THE FUNCTION, MAKE SURE
                THAT IN THE MATRIX STRUCT STANDS THE CORRECT DIM.       */
void vMatSub(Matrix *Mmat1, Matrix *Mmat2, Matrix *Msub)
{
   register int i,j;

   Msub->iNR = Mmat1->iNR;
   Msub->iNC = Mmat1->iNC;
   for (i=0; i<Mmat1->iNR; i++)
      for (j=0; j<Mmat1->iNC; j++)
         MAT(Msub, i, j) = MAT(Mmat1,i,j) - MAT(Mmat2,i,j);
}

/* vMatScMlt  : scalar multiplication a matrix
   input      : pointer to matrix
                scalar multiplier as a double
   output     : pointer resulting matrix
   return     :  - nothing -
   uses       : no functions but a macro in order to use the vectors
                as a (quadr.) matrix representations; advantage is the
                the unlimited size (!!) of the matrixes to be handled
                (this should be proper & portable C code)
   note       : NO ERROR CHECKING IS DONE IN THE FUNCTION, MAKE SURE
                THAT IN THE MATRIX STRUCT STANDS THE CORRECT DIM.       */
void vMatScMlt(Matrix *Mmat, double dMult, Matrix *Mprod)
{
   register int i,j;

   Mprod->iNR = Mmat->iNR;
   Mprod->iNC = Mmat->iNC;
   for (i=0; i<Mmat->iNR; i++)
      for (j=0; j<Mmat->iNC; j++)
         MAT(Mprod,i,j) = dMult * MAT(Mmat,i,j);
}

/* vMatScAdd  : scalar addition to each cell of a matrix
   input      : pointer to matrix
                scalar to be added as a double
   output     : pointer to resulting matrix
   return     :  - nothing -
   uses       : no functions but a macro in order to use the vectors
                as a (quadr.) matrix representations; advantage is the
                the unlimited size (!!) of the matrixes to be handled
                (this should be proper & portable C code)
   note       : NO ERROR CHECKING IS DONE IN THE FUNCTION, MAKE SURE
                THAT IN THE MATRIX STRUCT STANDS THE CORRECT DIM.       */
void vMatScAdd(Matrix *Mmat, double dAdd, Matrix *Mout)
{
   register int i,j;

   Mout->iNR = Mmat->iNR;
   Mout->iNC = Mmat->iNC;
   for (i=0; i<Mmat->iNR; i++)
      for (j=0; j<Mmat->iNC; j++)
         MAT(Mout,i,j) = dAdd + MAT(Mmat,i,j);
}

/* vScProd    : scalar product of two vectors
   input      : pointer to matrix1 (must be a (n*1) matrix, ie vector)
                pointer to matrix2 (idem)
   output     : scalar product of these two vectors
   return     :  - nothing -
   uses       : no functions but a macro in order to use the vectors
                as a (quadr.) matrix representations; advantage is the
```



```
                the unlimited size (!!) of the matrixes to be handled
                (this should be proper & portable C code)
   note       : NO ERROR CHECKING IS DONE IN THE FUNCTION, MAKE SURE
                THAT IN THE MATRIX STRUCT STANDS THE CORRECT DIM.       */
void vScProd(Matrix *Mmat1, Matrix *Mmat2, double *dRes)
{
   register int i;

   *dRes = 0.0;
   for (i=0; i<Mmat1->iNR; i++)
      *dRes += VEC(Mmat1,i) * VEC(Mmat2,i);
}
```

B.5 analyse — Measuring the Index Fund Performance

analyse provides the ex-post analysis of the tracking errors of all the portfolios that result from one GA experiment. analyse therefore read all the (in our case 100) min-files and prints out a table with all the tracking errors.

B.5.1 analyse.c

```
/************************************************************************
*                                                                       *
*   Program     GAPM - a Genetic Algorithm for Passive Management       *
*                                                                       *
*   File        analyse.c                                               *
*                                                                       *
*   Function    'analyse' calculates the ex-post analysis of the        *
*               tracking errors of the best GA individuals (that are    *
*               the portfolios) that in the min-x.suf files             *
*                                                                       *
*   Parameter   argc, *argv[]  for suffix of the GA run                 *
*                                                                       *
*   Caller      none - gets called form the system shell by the user    *
*                                                                       *
*   Uses        all the functions that eval() uses as well,i.e.         *
*               data handling functions from daxdata.c                  *
*               matrix functions from matrix.c                          *
*               vNoramlize() from eval.c                                *
*                                                                       *
*   Author      Dirk Eddelbuettel                                       *
*                                                                       *
*   Date        Mar 92 and Jul 92                                       *
*                                                                       *
************************************************************************/

#include "eval.h"                       /* include header file          */
#include "dax_data.h"                   /* share data type              */
#include "matrix.h"                     /* matrix types and macros      */
#include "master.h"                     /* parameter file types         */

#define LONG_LINE  1024                 /* min.xxx lines can be long !  */
#define SEPERATORS      " \t\v\n\r\f"   /* whitespaces for 'strtok'     */

void main(int argc, char *argv[])
{
   AllData Data;                        /* structure for all the data   */
   double *dVec;                        /* for data from min_file       */
   int *iSel;                           /* vector with selectors        */
   double *dWei;                        /* vector with weights          */
   double *dPhi;                        /* phis for art.index           */
```

```
double *dPFRet, *dIndRet;            /* vectors for return series    */
Matrix *mC, *mP;                     /* Matrix and vector for var    */
Matrix *mPhi, *mVec;                 /* tmp.vecs for const.var.part  */
double dToday, dYest;                /* for return calculations      */
double dEval;                        /* value from GAPM              */
double dConstVarPart;
int iGener, iNumEval;                /* Generat.+NumEval. in GAPM    */
double dTE1, dTE2, dTE3, dTE4, dTE5;
char cFilename[13];                  /* Filename variable            */
char cLongLine[LONG_LINE];
char *cWord;                         /* pointer to one word          */
FILE *fp;                            /* Filepointer for reading      */
register int i,j,k;                  /* loop variables               */



                                     /* Prototypes                   */
void vGetPhi(AllData *, double *, int);
int iGetMemForData(AllData *);
void vFreeData(AllData*);
int iReadData(AllData *, const char *, const char *);
void vMatProd(Matrix*, Matrix*, Matrix*);
void vMatScMlt(Matrix*, double, Matrix*);
void vScProd(Matrix *, Matrix *, double *);
double dStdErrDiff(double*, double*, int, int);
double dRMSEDiff(double*, double*, int, int);
void vCalcCovMat(AllData *, int, int, Matrix *);
void vNormalize(double*, int*, double*, int, int, int);
double sqrt(double);


if (argc != 2)
   {
   puts("ANALYSE: Please call with exactly one arguments,");
   puts("\t the suffix for the run that you want analyzed");
   exit(-1);
   }

Data.iNSh = 30;                      /* 30 shares in the DAX         */
Data.iObs = 247;                     /* 247 observations per company */

                                     /* Get parameters from master.x */
sprintf(cFilename, "master.%3.3s", argv[1]);
if ((fp = fopen(cFilename, "r")) == NULL)
   {
   fprintf(stderr, "Analyse: can't open %s for reading\n", cFilename);
   abort();
   }
fscanf(fp,MASTER_FORMAT,MASTER_VARS);/* actual reading, see master.h */
fclose(fp);                          /* for var. def. + declaration  */


if (iVersion != 2)                   /* basic consistency tests      */
   {
   fprintf(stderr, "Master: Parameter file is not matching\n");
   abort();
   }

if (iNormalizeSwitch == 1)           /* is it the convergence test ? */
   iNumSelectors = 30;               /* value was zero for GENESIS   */
                                     /* but we need 30 here          */

                                     /* get all the memory           */
MAKE_MATRIX(mC,Data.iNSh,Data.iNSh); /* covariance matrix            */
MAKE_MATRIX(mP,Data.iNSh,1);         /* var = xCx + px , this is p   */
MAKE_MATRIX(mPhi,Data.iNSh,1);       /* to determine p, we need phi  */
```



```
MAKE_MATRIX(mVec,Data.iNSh,1);       /* for const. var part          */
dVec      = (double*) calloc((size_t) iNumSelectors+iNumWeights,
                                        sizeof(double));
iSel      = (int*) calloc((size_t) iNumSelectors, sizeof(int));
dWei      = (double*) calloc((size_t) iNumWeights, sizeof(double));
dPhi      = (double*) calloc((size_t) Data.iNSh, sizeof(double));
dPFRet    = (double*) calloc((size_t) Data.iObs+1, sizeof(double));
dIndRet   = (double*) calloc((size_t) Data.iObs+1, sizeof(double));
if (  (iGetMemForData(&Data))        /* if memory problem in func.   */
       || !mC     || !mP      || !mPhi      || !mVec
       || !dVec   || !iSel    || !dWei      || !dPhi
       || !dPFRet || !dIndRet )
   {
   puts("ERROR, not enough memory for ANALYSIS !!");
   exit(-1);
   }

                                     /* read share data files        */
if (iReadData(&Data, PRICE_FILENAME, NUMBER_FILENAME))
   {
   fprintf(stderr,"Data incorrectly read !!\n");
   abort();
   }
                                     /* determine now C, phi and p   */
vCalcCovMat(&Data, 1, SERIES_LENGTH, mC);

vGetPhi(&Data,dPhi,SERIES_LENGTH);   /* get phis from function       */
for (i=0; i<Data.iNSh; i++)          /* copy phis into vector for    */
   VEC(mPhi,i) = dPhi[i];            /* all shares in index          */
free(dPhi);
vMatProd(mC, mPhi, mP);              /* 'p = C * phi'                */
vMatScMlt(mP, -2.0, mP);             /* p = p * (-2)                 */

                                     /* for all shares in index      */
                                     /* ie forall i,j elem I         */
vMatProd(mC,mPhi,mVec);              /* calc phi C phi               */
vScProd(mVec,mPhi,&dConstVarPart);

for (i=1; i<Data.iObs; i++)          /* calc. return on index        */
   dIndRet[i] = 100.0 * (Data.fInd[i] - Data.fInd[i-1])
                      / Data.fInd[i-1];

FREE_MATRIX(mPhi);
FREE_MATRIX(mVec);

puts("# Analysis of a GAPM run");
puts("# Output format from left to right:");
puts("#  - filenumber, eg min-1.abc is the 1st file from the abc run");
puts("#  - generation number of the dumped individual (=portfolio)");
puts("#  - evaluation number");
puts("#  - objective function value reported by GAPM");
puts("#  - recalculated function value for GAPM");
puts("# Thereafter follow the tracking errors between the index return");
puts("# and the portfolio returns for the periods");
puts("# 1-100 (in-sample) and 101-150, 151-200, 201-246 (out-of-sample)");

                                     /* Now everything ought to be   */
                                     /* ready, we are going to read  */
                                     /* as long as there are lines   */
                                     /* and process them one by one  */

for (k=1; k<=Totalexperiments; k++)  /* For all experiments          */
   {
                                     /* set min filename for exp. k  */
   if (Totalexperiments > 1)
      sprintf(cFilename, "min-%d.%3.3s", k, argv[1]);
```

```
else
   sprintf(cFilename, "min.%3.3s", argv[1]);

                                  /* Open file with best structs  */
if ((fp = fopen(cFilename, "r")) == NULL)
   {
   fprintf(stderr,"ERROR: Can't open file  %s\n",cFilename);
   continue;                      /* try next one                 */
   }
                                  /* read a line from the file    */
while (fgets(cLongLine, LONG_LINE, fp))
   {
                                  /* decompose line with 'strtok' */
   cWord = strtok(cLongLine, SEPERATORS);
   for(i=0; cWord != NULL; i++)   /* there was a word             */
      {
      if ((iNormalizeSwitch == 1) && (i<30))
         {
         dVec[i] = strtod(cWord, NULL);
         if (i==29)
            i+=30;
         }
      else
      if (i<iNumSelectors)                       /* selectors 1st*/
         dVec[i] = strtod(cWord, NULL);

      else if (i<iNumSelectors+iNumWeights)      /* weights 2nd  */
         dVec[i] = strtod(cWord, NULL);

      else if (i==iNumSelectors+iNumWeights)     /* best val.3rd */
         dEval = strtod(cWord, NULL);

      else if (i==iNumSelectors+iNumWeights+1)   /* gen.num.4th  */
         iGener = (int) strtol(cWord, NULL, 10);

      else if (i==iNumSelectors+iNumWeights+2)   /* eval num.5th */
         iNumEval = (int) strtol(cWord, NULL, 10);

      else                        /* there shouldn't be anything  */
         {
         fclose(fp);
         fprintf(stderr,"ANALYSE: Problem with line in %s\n",
                        cFilename);
         abort();
         }
                                  /* get next word out of the line*/
      cWord = strtok(NULL, SEPERATORS);
      }                           /* end of line analysis         */


                                  /* we have read + decomposed    */
                                  /* one line, now calculations   */

                                  /* normalize values             */
   vNormalize(dVec, iSel, dWei,
                 iNormalizeSwitch, iNumSelectors, iNumWeights);

   dToday = 0.0;                  /* calc. returns for that PF    */
   for (j=0; j<iNumWeights; j++)
      dToday += dWei[j] * Data.Sh[iSel[j]].fP[0];
   for (i=1; i<Data.iObs; i++)
      {
      dYest = dToday;             /* yesterday = val. of today    */
      dToday = 0.0;               /* set PfToday to zero          */
      for (j=0; j<iNumWeights; j++)  /* for all shares            */
         dToday += dWei[j] * Data.Sh[iSel[j]].fP[i];
```



```
                                  /* value, calc the PFreturn     */
         dPFRet[i]  = 100.0 * (dTody - dYest) / dYest;
         }

                                  /* var = xCx + phiCphi + px     */
      dTE1 = dConstVarPart;       /* set var to 'phi C phiOT      */
      for (i=0; i<iNumSelectors; i++)/* for all shares considered */
         {                        /* ie for all i,j elem. of P    */
                                  /* var = xCx + px               */
         for (j=0; j<iNumSelectors; j++)
            dTE1 +=   dWei[i] * dWei[j] * MAT(mC, iSel[i], iSel[j]);
         dTE1 += dWei[i] * VEC(mP,iSel[i]);
         }
      dTE1 = sqrt(dTE1);


      dTE2 = dStdErrDiff(dIndRet, dPFRet, 1, SERIES_LENGTH);

      dTE3 = dStdErrDiff(dIndRet, dPFRet, 101, 150);

      dTE4 = dStdErrDiff(dIndRet, dPFRet, 151, 200);

      dTE5 = dStdErrDiff(dIndRet, dPFRet, 201, 246);



                                  /* present results now          */
      printf("%3d%5d%6d%10.7f",k, iGener, iNumEval, dEval);
      printf("%10.7f%10.7f%10.7f", dTE1, dTE2, dTE3);
      printf("%10.7f%10.7f\n", dTE4, dTE5);

      }                           /* end of loop for lines        */
   fclose(fp);
   }                              /* end of loop for files        */
free(iSel); free(dWei);
free(dPhi); free(dPFRet);
free(dIndRet);
free(dVec);
FREE_MATRIX(mP);
FREE_MATRIX(mC);
vFreeData(&Data);
}
```

B.6 showpf — Print out the Portfolio Composition

This programme prints out a portfolio from a specified min-file.

B.6.1 showpf.c

```
/************************************************************************
*                                                                       *
*   Program     GAPM - a Genetic Algorithm for Passive Management       *
*                                                                       *
*   File        showpf.c                                                *
*                                                                       *
*   Function    'showpf' reads a user-specified min-file and normalises *
*               the portfolio values therein.                           *
*                                                                       *
*   Parameter   argc, *argv[]  for suffix                               *
*                                                                       *
*   Caller      none - gets called form the system shell by the user    *
```

```
*                                                                    *
*   Uses        vNormalize() from eval.c                             *
*                                                                    *
*   Author      Dirk Eddelbuettel                                    *
*                                                                    *
*   Date        Jul 92                                               *
*                                                                    *
**********************************************************************/

#include "eval.h"                       /* include header file         */
#include "master.h"                     /* parameter file types        */

#define LONG_LINE  1024                 /* min.xxx lines can be long !  */
#define SEPERATORS      " \t\v\n\r\f"   /* whitespaces for 'strtok'    */

void main(int argc, char *argv[])
{
   char cSuffix[4];
   int  iRepNum, iGenNum, iEvalNum;
   int  iErrFlag = 0;
   int  iEndFlag = 0;
   double *dVec;                        /* for data from min_file      */
   int *iSel;                           /* vector with selectors       */
   double *dWei;                        /* vector with weights         */
   double dEval;                        /* value from GAPM             */
   int iGener, iNumEval;                /* Generat.+NumEval. in GAPM   */
   char cFilename[13];                  /* Filename variable           */
   char cLongLine[LONG_LINE];
   char *cWord;                         /* pointer to one word         */
   FILE *fp;                            /* Filepointer for reading     */
   register int i,j,k;                  /* loop variables              */
   int iDummy;
   double dDummy;
   char cName[30][4] = { "ALV", "BAS", "BAY", "BHW", "BMW",
                      "BVM", "CBK", "CON", "DAI", "DGS",
                      "DBC", "DBK", "DRB", "HEN", "HFA",
                      "KAR", "KFH", "LIN", "LHA", "MAN",
                      "MMW", "MET", "PRS", "RWE", "SCH",
                      "SIE", "THY", "VEB", "VIA", "VOW" };

                                        /* Prototypes                  */
   void vNormalize(double*, int*, double*, int, int, int);

   if (argc != 5)
      iErrFlag = 1;
   else
      iErrFlag = ! (   sprintf(cSuffix, "%3.3s", argv[1]  )
                   &&  sscanf(argv[2], "%d",    &iRepNum )
                   &&  sscanf(argv[3], "%d",    &iGenNum )
                   &&  sscanf(argv[4], "%d",    &iEvalNum) );

   if (iErrFlag)
      {
      puts("SHOWPF: Please call with exactly four arguments,");
      puts("\t the suffix for the run");
      puts("\t the repetition number within the run");
      puts("\t the generation number");
      puts("\t the evaluation number");
      exit(-1);
      }

                                        /* Get parameters from master.x */
   sprintf(cFilename, "master.%3.3s", argv[1]);
   if ((fp = fopen(cFilename, "r")) == NULL)
      {
      fprintf(stderr, "ShowPF: can't open %s for reading\n", cFilename);
```



```
   abort();
   }
fscanf(fp,MASTER_FORMAT,MASTER_VARS);/* actual reading, see master.h */
fclose(fp);                          /* for var. def. + declaration  */

if (iVersion != 2)                   /* basic consistency tests      */
   {
   fprintf(stderr, "ShowPf: Parameter file is not matching\n");
   abort();
   }

if (iNormalizeSwitch == 1)           /* is it the convergence test ? */
   iNumSelectors = 30;               /* value was zero for GENESIS   */
                                     /* but we need 30 here          */

                                     /* get all the memory           */
dVec      = (double*) calloc((size_t) iNumSelectors+iNumWeights,
                                             sizeof(double));
iSel      = (int*) calloc((size_t) iNumSelectors, sizeof(int));
dWei      = (double*) calloc((size_t) iNumWeights, sizeof(double));
if (!dVec || !iSel || !dWei)         /* if memory problem in func.   */
   {
   puts("ERROR, not enough memory for SHOWPF !!");
   exit(-1);
   }

                                     /* Open file with best structs  */
if (Totalexperiments > 1)
   sprintf(cFilename, "min-%d.%s", iRepNum, cSuffix);
else
   sprintf(cFilename, "min.%s", cSuffix);
if ((fp = fopen(cFilename, "r")) == NULL)
   {
   fprintf(stderr,"ShowPF: Can't open file  %s\n",cFilename);
   abort();
   }

                                     /* Now everything ought to be   */
                                     /* ready, we are going to read  */
                                     /* as long as there are lines   */
                                     /* and process them one by one  */
                                     /* read a line from the file    */
while (  (!iEndFlag)
     &&(fgets(cLongLine, LONG_LINE, fp)))
   {
                                     /* decompose line with 'strtok' */
   cWord = strtok(cLongLine, SEPERATORS);
   for(i=0; cWord != NULL; i++)      /* there was a word             */
      {
      if ((iNormalizeSwitch == 1) && (i<30))
         {
         dVec[i] = strtod(cWord, NULL);
         if (i==29)
            i+=30;
         }
      else if (i<iNumSelectors)                    /* selectors 1st*/
         dVec[i] = strtod(cWord, NULL);

      else if (i<iNumSelectors+iNumWeights)  /* weights 2nd  */
         dVec[i] = strtod(cWord, NULL);

      else if (i==iNumSelectors+iNumWeights) /* best val.3rd */
         dEval = strtod(cWord, NULL);

      else if (i==iNumSelectors+iNumWeights+1)    /* gen.num.4th  */
         iGener = (int) strtol(cWord, NULL, 10);
```

```
      else if (i==iNumSelectors+iNumWeights+2)    /* eval num.5th */
         iNumEval = (int) strtol(cWord, NULL, 10);

      else                   /* there shouldn't be anything  */
         {
         fclose(fp);
         fprintf(stderr,"ShowPF: Problem with line in %s\n", cFilename);
         abort();
         }
                                    /* get next word out of the line*/
      cWord = strtok(NULL, SEPERATORS);
      }                             /* end of line analysis         */

   if (  (iGener   == iGenNum )
     &&(iNumEval == iEvalNum))
      iEndFlag = 1;                 /* found the one we looked for  */

   }                                /* end of loop for lines        */
fclose(fp);

if (!iEndFlag)
   {
   printf("ShowPF: In file %s paramters %d and %d not found\n",
                cFilename, iGenNum, iEvalNum);
   abort();
   }

                                    /* normalize values             */
vNormalize(dVec, iSel, dWei,
                       iNormalizeSwitch, iNumSelectors, iNumWeights);

                                    /* Shell's sorting for sel's    */
for (i=iNumSelectors/2; i>0; i/=2)
   for (j=i; j<iNumSelectors; j++)
      for (k=j-i; k>= 0; k-=i)
         if (iSel[k] > iSel[k+i])
            {
            iDummy    = iSel[k];
            dDummy    = dWei[k];
            iSel[k]   = iSel[k+i];
            dWei[k]   = dWei[k+i];
            iSel[k+i] = iDummy;
            dWei[k+i] = dDummy;
            }

                                    /* Beautify: erase shares with  */
                                    /* dWei = 0 (where 30 or 31 was */
                                    /* selector) and collect several*/
                                    /* entries to one share         */

iEndFlag = iNumSelectors;           /* memorize number of shares    */
for (i = 0; i < iEndFlag-1; )       /* for all but the last         */
   if (iSel[i] == iSel[i+1])        /* if two times the same        */
      {
      iEndFlag--;                   /* there is one less            */
      dWei[i] += dWei[i+1];         /* collect volume               */
      for (j=i+1; j<iEndFlag; j++)
         {
         iSel[j] = iSel[j+1];
         dWei[j] = dWei[j+1];
         }
      }
   else
      i++;
```



```
for (i=0; i<iEndFlag; i++)          /* for all shares               */
   if (dWei[i] == 0.0)              /* if no volume                 */
      {
      iEndFlag--;                   /* there is one less            */
      for (j=i; j<iEndFlag; j++)
         {
         iSel[j] = iSel[j+1];
         dWei[j] = dWei[j+1];
         }
      }
                                    /* Output                       */
printf("# Portfolio from %s\n", cFilename);
printf("# from generation %d, evaluation %d with value %lf\n",
                                       iGener, iNumEval, dEval);
printf("# Setup: %d selectors, %d weights, ", iNumSelectors, iNumWeights);
printf("%d effectively selected\n",iEndFlag);
for (i=0,j=0; i<iEndFlag; i++,j++)
   {
   printf("%3.3s %2.1d %10.6lf", cName[iSel[i]], iSel[i], 100.0*dWei[i]);
   if (j == 3)
      {
      printf("\n");
      j=-1;
      }
   else
      printf("   ");
   }
printf("\n");

free(iSel); free(dWei); free(dVec);
}
```

B.7 testpf — Test the Portfolio Mean

This programme does perform a t-test for the null hypothesis of equal means. It does read the analyse-file containing all the tracking errors for a run; the user provides a value to compare with (e.g. the OLS portfolio tracking error) and the programme gives the test result.

B.7.1 testpf.c

```
/************************************************************************
*                                                                       *
*   Program     GAPM - a Genetic Algorithm for Passive Management       *
*                                                                       *
*   File        testpf.c                                                *
*                                                                       *
*   Function    'testpf' t-tests a portfolio tracking error against a   *
*               user given value (eg OLS estimated one)                 *
*                                                                       *
*   Parameter   argc, *argv[]  for suffix                               *
*                                                                       *
*   Caller      none - gets called form the system shell by the user    *
*                                                                       *
*   Uses        nothing apart from standard C functions and the         *
*               normal law proxi from the end of this file              *
*                                                                       *
*   Author      Dirk Eddelbuettel                                       *
*                                                                       *
*   Date        Jul 92                                                  *
*                                                                       *
************************************************************************/
```

```
#include "master.h"

#define LONG_LINE  1024                 /* min.xxx lines can be long !  */
#define PROGNAME   "TestPF"

void main(int argc, char *argv[])
{
   char cSuffix[4];
   int  iTErrNum;
   double dRefVal;
   int  iErrFlag = 0;
   float *fVec;                         /* for data from min_file       */
   double dAverage = 0;
   double dStdErr = 0;
   double dNumLin;
   double dGauss;
   char cFilename[13];                  /* Filename variable            */
   char cLongLine[LONG_LINE];
   FILE *fp;                            /* Filepointer for reading      */
   register int i;                      /* loop variables               */

   double sqrt(double);
   double stdnormv(double);

   if (argc != 4)
      iErrFlag = 1;
   else
      iErrFlag = ! (   sprintf(cSuffix, "%3.3s", argv[1]  )
                    && sscanf(argv[2], "%d",     &iTErrNum)
                    && sscanf(argv[3], "%lf",    &dRefVal ) );

   if (iErrFlag)
      {
      puts("TestPF: Please call with exactly three arguments,");
      puts("\t the suffix for the run");
      puts("\t the number of the tracking error to be tested,");
      puts("\t the reference value (eg. OLS-TE)");
      exit(-1);
      }

   if ((iTErrNum < 1) || (iTErrNum > 4))
      {
      puts("TestPF: tracking error number must be between 1 and 4");
      exit(-1);
      }

                                        /* Get parameters from master.x */
   sprintf(cFilename, "master.%3.3s", argv[1]);
   if ((fp = fopen(cFilename, "r")) == NULL)
      {
      fprintf(stderr, "%s: can't open %s for reading\n",
                         PROGNAME, cFilename);
      abort();
      }
   fscanf(fp,MASTER_FORMAT,MASTER_VARS);/* actual reading, see master.h */
   fclose(fp);                          /* for var. def. + declaration  */
                                        /* there are Totalexp*Savesize  */
                                        /* lines in analyse.<suffix>    */
   dNumLin = (double) Totalexperiments * Savesize;


   if (iVersion != 2)                   /* basic consistency tests      */
      {
      fprintf(stderr, "%s: Parameter file is not matching\n", PROGNAME);
```



```
      abort();
      }

   fVec      = (float*) calloc((size_t) dNumLin, sizeof(float));
   if (!fVec)                           /* if memory problem            */
      {
      puts("ERROR, not enough memory for TESTPF !!");
      exit(-1);
      }

                                        /* Open file                    */
   sprintf(cFilename, "analyse.%s", cSuffix);
   if ((fp = fopen(cFilename, "r")) == NULL)
      {
      fprintf(stderr,"%s: Can't open file  %s\n", PROGNAME, cFilename);
      abort();
      }

                                        /* Now everything ought to be   */
                                        /* ready, we are going to read  */
                                        /* as long as there are lines   */
                                        /* and process them one by one  */
                                        /* read a line from the file    */
   i = 0;
   while (fgets(cLongLine, LONG_LINE, fp))
      if (   (cLongLine[0] != '#' )     /* not a comment line           */
          && (cLongLine[0] != '\n') )   /* not an empty line            */
         {
         switch (iTErrNum)
            {
            case 1: sscanf(cLongLine,
                           "%*d %*d %*d %f %*f %*f %*f %*f %*f",
                           &fVec[i]);
                    break;
            case 2: sscanf(cLongLine,
                           "%*d %*d %*d %*f %*f %*f %f %*f %*f",
                           &fVec[i]);
                    break;
            case 3: sscanf(cLongLine,
                           "%*d %*d %*d %*f %*f %*f %*f %f %*f",
                           &fVec[i]);
                    break;
            case 4: sscanf(cLongLine,
                           "%*d %*d %*d %*f %*f %*f %*f %*f %f",
                           &fVec[i]);
                    break;
            default:break;              /* there should be no default   */
                                        /* as we've tested iTErrNum     */
            }
         dAverage += (double) fVec[i++];
         }                              /* end if valid line            */
   fclose(fp);

   if (i != (int) dNumLin)
      puts("TestPF: bad number of lines in file!");
   else
      {
      dAverage /= dNumLin;
      for (i = 0; i < (int) dNumLin; i++)
         dStdErr +=   ((double)fVec[i] - dAverage)
                    * ((double)fVec[i] - dAverage);
      dStdErr = sqrt(dStdErr / (dNumLin - 1));
      dGauss = (dAverage - dRefVal) / dStdErr * sqrt(dNumLin);
      if (dAverage > dGauss)
         printf("Reject!! Average %lf (StdErr %lf) Gauss %lf P %lf\n",
                        dAverage, dStdErr, dGauss, stdnormv(dGauss));
```

```
      else
         printf("Accept   Average %lf (StdErr %lf) Gauss %lf P %lf\n",
                        dAverage, dStdErr, dGauss, stdnormv(dGauss));

      }

   free(fVec);
}

double stdnormv(double d)              /* cumulated standard normal
                                          distributed approximation from
                                          Abramowitz,M: Handbook of
                                          Mathematical Functions (1965)*/
{
   double p, s = 0;

   p = 1 / ( 1 + 0.2316419 * d);

   s = 0.31938153 * p;
   s = s + (-0.356563782 * p * p);
   s = s + ( 1.781477937 * p * p * p);
   s = s + (-1.821255978 * p * p * p * p);
   s = s + ( 1.330274429 * p * p * p * p * p);

   s = s * ( 1 / sqrt(2*3.141592654) ) * exp(-d * d / 2);

   return(1 - s);
}
```

B.8 Changes made to GENESIS

As we wanted to use the GENESIS package as a black box that provides genetic algorithms operations, we did not want to change its internal operations. Nevertheless, we undertook some minor changes that concern rather the in- and output operations than the genetic operations at the heart of GENESIS. We will now show the changes that we made and of course not print out the complete source code of GENESIS, but only the changed lines.

The first change regards the file `init.c` that contains the instructions for setting up the initial population. Because the MS-DOS operating system does not allow the name kind of filenames, we had to change as it is shown by the following lines.

```
        if (Totalexperiments > 1)
                /*     edd 29.07.92 Replaced the following line
                       sprintf(Bestfile, "%s.%d", Minfile, Experiment+1);
                       by */
                {
                sprintf(Bestfile, "min-%d.", Experiment+1); /* min-<n>. */
                i = strlen(Bestfile);
                Bestfile[i]   = Minfile[4];
                Bestfile[i+1] = Minfile[5];
                Bestfile[i+2] = Minfile[6];
                Bestfile[i+3] = '\0';
                }
                /*     in order to permit dos-compatible filename of the
                       'Bestfile', we use min-<n>.<xyz> instead of
                       min.<xyz>.<n> where <xyz> is the suffix
                       (eg 'x20') and <n> the number of the experiment.
                       For the (general) case where Totalexperiments=1,
                       Minfile is set to min.<abc> by input.c . We
                       pick thus the suffix <abc> out of Minfile         */
```


The following change ensures that the data describing the best structures in a population is always dumped into an empty file and not attached to an pre-existing file, thus eventually mixing data from several runs.

```
        /* edd 11.12.91   changed 'append-mode' to write mode, thus a new
                          new file is always started from scratch        */
        fp = fopen(Bestfile, "w");
```

The next change concerns the file `measure.c` that contains the instruction for the performance measurements and displays.

```
                /*
                 *       edd 03.12.91
                 *
                 *     changed this line in order the show the
                 * performance more precisely
                 *
                 *         printw("Performance: %0.4f ", New[Best_guy].Perf);
                 *                                 !
                 *       new value:           %0.8f
                 */
                printw("Performance: %0.8f ", New[Best_guy].Perf);
```

The last change concerns the file `report.c` that builds the convergence report from the data in the `out`-file. We simply allowed for far more lines resulting from longer than usual experiments.

```
#define ROWS 1000                /* edd changed from 200 to 1000       */
#define COLUMNS 9

double average[ROWS][COLUMNS];
double variance[ROWS][COLUMNS];
double line[COLUMNS];
```

References

Baker, J.E. (1987). *Reducing bias and inefficiency in the selection algorithm*, in Grefenstette (1987a), pages 14–21.

Banz, R.W. (1981). "The Relationship between Return and Market Value of Common Stock," *Journal of Financial Economics*, **6**, pages 3–18.

Belew, R.K. and L.B. Booker, editors (1991). *Proceedings of the fourth international conference on genetic algorithms* held at the University of California, San Diego, July 13-16, 1991. Morgan Kaufman Publishers, San Mateo.

Bethke, A.D. (1981). *Genetic algorithms as functions optimisers*, Doctoral dissertation, University of Michigan, Dissertation Abstracts International 41(9), 3503B.

Bishop, E. (1990a). *Indexation*, Euromoney Publications PLC, London.

Bishop, E. (1990b). *An introduction to indexation*, chapter 1 in Bishop (1990a), pages 1–18.

Bollerslev, T. (1986). "Generalized autoregressive conditional heteroscedasticity," *Journal of Econometrics* **37**, pages 307–327.

Bridges, C.L. and D.E. Goldberg (1987). *An analysis of reproduction and crossover in a binary-coded genetic algortithm*, in Grefenstette (1987a), pages 9–13.

Camus, A. (1947). *La peste*, Editions Gallimard, Paris.

Caruana, R.A. and J.D. Schaffer (1988). "Representation and hidden bias: Gray vs. binary coding for genetic algorithms,". In J. Laird, editor, *Proceedings of the fifth International Conference on Machine Learning*, Morgan Kaufman, San Mateo.

Chamberlain, G. (1983). "Funds, factors, and diversification in arbitrage pricing models," *Econometrica*, **51**, pages 1305–1323.

Chamberlain, G. and M. Rothschild (1983). "Arbitrage, factor structure, and mean-variance analysis on large asset markets," *Econometrica*, **51**, pages 1281–1304.

Davidor, Y. (1991). *Genetic algorithms and robotics: a heuristic strategy for optimization*, World Scientific Publishing, Singapore.

Davis, L., editor (1987). *Genetic algorithms and simulated annealing*, Morgan Kaufman, San Mateo.

Davis, L. (1989). *Adapting operator probabilities in genetic algorithms*, in Schaffer (1989), pages 61–69.

Davis, L., editor (1991). *Handbook of genetic algorithms*, Van Nostrand Reinhold, New York.

Davis, L. and M. Steenstrup (1987). *Genetic algorithms and simulated annealing: an overview*, chapter 1 in Davis (1987), pages 1–11.

De Jong, K.A. (1975). *An analysis of the behavior of a class of genetic adaptive systems*, Doctoral dissertation, University of Michigan, Dissertation Abstracts International 36(10), 5140B.

Eiben, A.E. , E.H.L. Aarts and K.M. Van Hee (1990). *Global convergence of genetic algorithms: a Markov chain analysis*, in Schwefel and Männer (1990), pages 4–12.

Elton, E.J. and M.J. Gruber (1984). *Modern portfolio theory and investment analysis*, Second edition, John Wiley, New York.

Eshelman, L.J. and J.D. Schaffer (1991). *Preventing premature convergence in genetic algorithms by preventing incest*, in Belew and Booker (1991), pages 115–122.

Engesser, H. (1989). *Duden Informatik*, Bibliographisches Institut, Mannheim.

Engle, R.F. (1982). "Autoregressive conditional heteroscedasticity with estimates of the variance of United Kingdom inflation," *Econometrica*, **50**, pages 987–1007.

Fama, E.F. (1970). "Efficient capital markets: a review of theory and empirical work," *Journal of Finance*, **25**, pages 383–417.

Fama, E.F. (1976). *Foundations of finance*, Basic Books, New York.

Fama, E.F. (1991). "Efficient capital markets: II," *Journal of Finance*, **46**, pages 1575–1617.

Fama, E. and M. Miller (1972). *Theory of finance*, Holt, Rinehart and Winston, New York.

Frankfurter Wertpapierbörse AG (1990). *Deutscher Aktienindex — DAX*, information paper from the Frankfurter Werpapierbörse AG, 1990.

Goldberg, D.E. (1987). *Simple genetic algorithms and the minimal, deceptive problem*, chapter 6 in Davis (1987), pages 74–88.

Goldberg, D.E. (1989a). *Genetic algorithms in search, optimization, and machine learning*, Addison-Wesley, Reading.

Goldberg, D.E. (1989b). "Genetic algorithms and Walsh functions: Part I, a gentle introduction" and "Genetic algorithms and Walsh functions: Part II, deception ans its analysis," *Complex Systems*, **3**, pages 129–152 and 153–171.

Goldberg, D.E. (1989c). *Sizing populations for serial and parallel genetic algorithms*, in Schaffer (1989), pages 70–79.

Goldberg, D.E. (1989d). *Zen and the art of genetic algorithms*, in Schaffer (1989), pages 80–85.

Goldberg, D.E. and J.T. Richardson (1987). *Genetic algorithms with sharing for multimodal function optimization*, in Grefenstette (1987a), pages 41–49.

Göppl, H. (1990). *Kapitaltheorie*, Mimeo, Universität Karlsruhe.

Grefenstette, J.J., editor (1985). *Proceedings of the first international conference on genetic algorithms and their application*, Lawrence Erlbaum Associates, New Jersey.

Grefenstette, J.J. (1986). "Optimization of control parameters for genetic algorithms," *IEEE Transactions on Systems, Man, and Cybernetics*, **SMC-16**, pages 122–128.

Grefenstette, J.J., editor (1987a). *Genetic algorithms and their applications: Proceedings of the second International Conference on Genetic Algorithms* held at the Massachusetts Institute of Technology, Cambridge, July 28-31, 1987. Lawrence Erlbaum Associates, New Jersey.

Grefenstette, J.J. (1987b). *Incorporating problem specific knowledge into genetic algorithms*, chapter 4 in Davis (1987), pages 43–60.

Grefenstette, J.J. and J.E. Baker (1989). *How genetic algorithms work: a critical look at implicit parallelism*, in Schaffer (1989), pages 20–27.

Grefenstette, J.J. , L. Davis and D. Cerys (1991). *GENESIS and OOGA: two genetic algorithm systems*, The Software Partnership, Melrose.

Holland, J.H. (1973). "Genetic algoritms and the optimal allocation of trials," *SIAM Journal of Computing*, **2**, pages 888-105.

Holland, J.H. (1975). *Adaption in natural and artificial systems*, The University of Michigan Press, Ann Arbor.

Holland, J.H. (1987). *Genetic algorithms and classifier systems: Foundations and future directions*, in Grefenstette (1987a), pages 82–89.

Jog, P., J.Y. Suh and D. Van Gucht (1989). *The effects of population size, heuristic crossover and local improvement on a genetic algorithm for the travelling salesman problem*, in Schaffer (1989), pages 110–116.

Kernighan, B.W. and D.M. Ritchie (1988). *The C programming language, second edition, ANSI C*, Prentice-Hall, Englewood Cliffs, NJ.

Knuth, D.E. (1987). "`plain.tex`," TEX typesetting system, Public domain software.

Kramer, R. and M. Morlock (1988). *Wissensbasierte System im Operations Research I*, Technical Report WIOR-327, Universität Karlsruhe.

Kreps, D.M. (1990). *A course in microeconomic theory*, Harvester Wheatsheaf, New York.

Lennon, J. and P. McCartney (1968). "Revolution 1," *The White Album*, EMI Records, London.

Liepins, G.E., M.R. Hilliard, M.R. Palmer and M. Morrow (1987). *Greedy Genetics*, in Grefenstette (1987a), pages 90–99.

Liepins, G.E. and M.R. Hilliard (1989). "Genetic algorithms: foundations and applications," *Annals of Operations Research*, **21**, pages 31–58.

Liesching, R. and V. Manchanda (1990). *Establishing the fund*, chapter 7 in Bishop (1990a), pages 93–114.

Lintner, J. (1965). "The valuation of risk assets and the selection of risky investments in stock portfolios and capital budgets," *Review of Economics and Statitics*, **47**, pages 13–37.

Loistl, O. (1991). *Kapitalmarkttheorie*, R. Oldenbourg, München.

Markowitz, H.M. (1952). "Portfolio selection," *Journal of Finance*, **7**, pages 77–91.

Markowitz, H.M. (1990). Foreword to Bishop (1990a).

Mattes, E. (1992). "`emtex386.doc`," EMTEX386 programme documentation, Public domain software.

Meade, N. and G.R. Salkin (1989). "Index Funds — construction and peformance measurements," *Journal of the Operational Research Society*, **40**, Number 10, pages 871–879.

Meade, N. and G.R. Salkin (1990). "Developing and maintaining an equity index fund," *Journal of the Operational Research Society*, **41**, Number 7, pages 599–607.

Michalewicz, Z. and C.Z. Janikow (1991). *Handling constraints in genetic algorithms*, in Belew and Booker (1991), pages 151–157.

Mossin, J. (1966). "Equilibrium in a capital asset market," *Econometrica*, **34**, pages 768–783.

Pao, Y.-H. (1989). *Adapative Pattern Recognition and Neural Networks*, Addison-Wesley, Reading.

Richardson, J.T, M.R. Palmer, G.E. Liepins and M.R. Hilliard (1989). *Some guidelines for genetic algorithms with penalty functions*, in Schaffer (1989), pages 191–197.

Richter, C. (1988). *Optimierungsverfahren und BASIC-Programme*, Akademie Verlag, Ost-Berlin.

Richter, H.-J. (1990). *DAX — the concept behind the DAX Performance Index*, Deutsche Bank Group (Quantitative Research), Frankfurt.

Schaffer, J.D., editor (1989). *Proceedings of the third International Conference on Genetic Algorithms* held at the George Mason University, June 4-7 1989. Morgan Kaufman Publishers, San Mateo.

Schaffer, J.D., R.A. Caruana, L.J. Eshelman and R. Das (1989). *A study of control parameters affecting online performance of genetic algorithms for function optimization*, in Schaffer (1989), pages 51–60.

Schwefel, H.-P. and R. Männer, editors (1990). *Parallel problem solving from nature*, Lecture Notes in Computer Sciences, Number 496, Springer Verlag, Berlin.

Sharpe, W.F. (1964). "Capital asset prices: a theory of market equilibrium under conditions of risk," *Journal of Finance*, **19**, pages 425–442.

Suh, J.Y. and D. Van Gucht (1987). *Incorporating heuristic information into genetic search*, in Grefenstette (1987a), pages 100–107.

Syswerda, G. (1989). *Uniform crossover in genetic algorithms*, in Schaffer (1989), pages 2–9.

Taylor, S.J. (1989). "Simulating financial prices," *Journal of the Operational Research Society* **40**, pages 567–569.

Tomlinson, L. (1990). *Indices*, chapter 2 in Bishop (1990a), pages 19–36.

Whitley, D. (1989). *The Genitor algorithm and selective pressure: why rank-based allocation of reproductive trials is best*, in Schaffer (1989), pages 116–121.