University of Turin

School of Management and Economics

Master's Degree in
Quantitative Finance and Insurance

Portfolio optimization through genetic algorithms in an artificial stock market

Author:
Andrea Chiamenti

Supervisor:
prof. Pietro Terna

Opponent:
prof. Sergio Margarita

March 2015

Contents

Acknowledgements

I would like to thank my supervisor, professor Pietro Terna, for his constant presence. His advices, his support and his enthusiasm played a big role in the realization of my work.

I also have to thank all my fellow graduate students who, with their experiences and their passion, helped me get through my studies.

I want then to thank my small family, that does not need words to show me love, and all my friends for always being with me.

Special thanks to five great ladies and guys, for having been my second family.

Thanks to Sonia, for living life with me.

And thanks to my incredible mother, whose trust in me has never been shaken and whose strength made all my achievements possible.

Introduction

The selection of the optimal portfolio is a widely discussed and massively studied topic. Besides the theoretical interest, succeeding in selecting the best and most efficient combination of financial assets has heavy practical effects, ranging from large profits (or losses) for institutional investors to good (or bad) wealth management for households.

Since the 1950s, when Markowitz developed his portfolio selection theory, the field witnessed large achievements. The basic mean-variance approach has been explored and expanded, leading to a vast set of different portfolio construction approaches. Many criticisms dealt with the assumptions of Markowitz's model: asset returns, for instance, have been observed to not follow a Gaussian (and not even a symmetric) distribution. Practical problems in the application of modern portfolio theory, indeed, led Black and Litterman to expand the model, accounting for investors views.

Considering instead the field of asset pricing, the Capital Asset Pricing Model has been an important starting point for portfolio strategies development: its anomalies, indeed, spurred researchers to analyse stocks behaviour under particular circumstances. What emerged, resulted in the Fama and French's three-factor model, then expanded by Carhart with the introduction of the momentum analysis.

During the same years, starting from Modigliani's work, the Life Cycle theory developed: households life was now sectioned and analysed, thus to understand wealth fluctuations individuals have to face, and adjust financial choices accordingly to reach better performances. This, of course, influenced portfolio theory as well: new goals, such as optimizing life cycle consumption, were now to be achieved by assets trading.

In the end what is evident, even from a brief description of the facts, is that studies about optimal portfolio composition are in constant development, always confronting with reality and real investors needs.

At the same time, the second half of the past century brought an impressive development in computer science. In particular, increased computational power and more sophisticated programming knowledge boosted the field of heuristic techniques, a practical tool allowing to find optimal solutions whenever the search space is particularly large. When, in the 1970s,

Holland combined them with the biological theory of evolution, the world assisted to the birth of an optimization method known as *genetic algorithms*. Treating the candidate solutions as biological individuals, such algorithms apply genetic evolutionary principles in order to mutate and combine the possible solutions and develop a new, better fitting generation of results.

Genetic algorithms, hence, appear to meet the needs of financial markets, where traders have to deal with enormous amounts of data and, among them, have to come up with an optimal solution.

This work moves from the interest and curiosity about the above mentioned fields and tries to design a computer application of genetic algorithms to portfolio selection. The first step of the research is to build, by means of Python's programming language, an artificial stock market inside which different portfolio strategies can be run, observed and tested. A real financial market is an extremely complex entity due, in particular, to the nature of stocks: what influences their prices ranges from corporate choices to international dynamics. My market, hence, represents a simplified reproduction of the reality, where prices vary randomly. The second step consists in investigating Python's genetic algorithms possibilities, offered by the library pyevolve, to design an efficient stocks selection approach.

In order to allow a full understanding of the core of the experiments, the first chapters are dedicated to the introduction and explanation of the main concepts on which the work is based.

Chapter 1 presents a brief history of genetic algorithms development, before illustrating how the technique works: the role of its main elements, such as fitness function, selection methods and genetic operators, is illustrated in detail.

In chapter 2, the basic knowledge about portfolio selection can be found: an explanation about the role of risk, of diversification and of utility functions introduces a dissertation about the mean-variance approach. The topic is much wider than what is talked about in the chapter, but to the ends of this work it is enough to understand the theory behind it.

Chapter 3 deals with one of the main instruments needed to reach my goal: Python's library pyevolve. Its functioning is explained by means of a practical tutorial, using small sample programs to highlight the main features of the library and the possibilities it offers.

In chapter 4 I present an interesting application which, however, goes beyond the purpose of this work and is hence only used as a link between plain pyevolve and its application to stock portfolios.

Chapter 5 illustrates in detail how the artificial stock market is constructed, carefully explaining the code meaning and its logic. There, all the different portfolio strategies are also set up.

Chapter 6 reports the results obtained during the market simulations.

The attention there is focused on two different aspects: on one hand, various prices dynamics are tested; on the other hand, the effects of alternative genetic algorithms settings are observed.

Chapter 7, finally, further expands the analysis of the results and concludes the writing by recalling the fundamental ideas highlighted during the work. This allows to have a good overall vision of the concepts, useful for a reasoning on future developments and applications which, speaking about genetic algorithms, are definitely vast.

Chapter 1

Genetic algorithms

Genetic algorithms are one of the most important parts of a wider set of computer science instruments known as *evolutionary computation*. As the name suggests, such tools are constructed based on the Darwinian evolutionary principles observed in nature: the same ideas are applied and the same patterns are mimicked in order to improve and specialize the individuals according to the specific problem being faced. The field of application for evolutionary computation (and for all its subsets) is the one of search problems and optimization, with particular attention to the so-called black box problems and, in general, to all those cases in which the solver has no idea about how a solution would look like.

More specifically, genetic algorithms belong to the subgroup of *evolutionary algorithms*. Those algorithms, genetic ones included, can be defined as heuristic (or metaheuristic) optimization methods. With the term *heuristic* we denote a whole set of strategies devoted to problem solving, applied to cases in which a complete search in the entire space of feasible solutions would be impossible, or even just not efficient. This choice, of course, sacrifices completeness, so the risk is that of finding a solution which is not optimal. On the other hand, the resulting process will turn out to be more rapid and the result will still be widely acceptable. Famous heuristic method examples are the well-known rule of thumb, trial and error, stereotyping and even common sense.

Genetic algorithms are constructed applying biological evolutionary principles such as organisms reproduction, natural selection, genes mutation and recombination and genetic inheritance. The first step is to randomly create an initial population from the set of possible values admitted as a solution. This first generation is tested by means of a fitness function, which must always be present in order to constantly determine the adequateness of the candidate solutions, somehow mimicking the way nature tests organisms to find out which ones are more suitable to survive. At this point, a second generation arises from the application of the above-mentioned genetic pro-

cesses to the most fit individuals, that will have their features mutated, cross-combined and bequeathed. The whole process is then iterated until a predetermined level of fitness is reached by an individual or until a given number of generations is created.

1.1 A brief history

The application of Darwinian concepts to computational science and, in particular, to computer science is a process that originated at the very beginning of the 1950s. One of the earliest works in this field comes, not surprisingly, from the father of computer science: Alan Turing (1950), in his paper *Computing machinery and intelligence*, develops an embryonic application of evolution to an artificial intelligence. The main idea was to develop a computer so capable in holding a conversation that a person could not distinguish it from a human being, arising the possibility of defining such machine as "intelligent" (this is the so-called *Turing test*). The work is far from what we define to be genetic algorithms nowadays but, still, introduced the evolutionary field in computer science: rather than creating a machine on the base of an adult's mind, indeed, Turing decided to program an artificial child's intelligence, able to evolve with conversational experience.

A few years later, the Norwegian-Italian psychologist and mathematician Barricelli (1954) published, in the journal *Methoods*, a pioneering work in the artificial evolution field, *Esempi numerici di processi di evoluzione*, which unfortunately was not highly noticed by the scientific community at that time. The paper illustrated a series of experiments held at the Institute for Advanced Study in Princeton, New Jersey, in 1953, and has been republished in English in 1957. After a visit at the Princeton University in 1954, Barricelli also started to develop a chess-playing program based on evolutionary principles.

Another early and innovative work was published by the Australian quantitative geneticist Fraser (1957): *Simulation of genetic systems by automatic digital computers*. The relevance of this paper is high since Frasier's programs, which simulated natural selection, encapsulated most of the features observable in modern genetic algorithms.

This same credit is to be assigned to Bremermann (1962), a German-American biophysicist and mathematician, whose work approached the search of optimal solutions in a modern way, through the application of strictly genetic operations such as mutations, crossovers and recombination.

That was a period of ferment in the field, and evolutionary principles found application to many different areas. Box (1957) developed the so-called *Evolutionary Operation* (EVOP), a process aimed at maximizing and optimizing manufacturing production through an experience-based progression: as long as the output is satisfactory, EVOP is able to extrapolate data

in order to make slight changes that will improve the manufacturing efficiency. Although born as an industrial application, EVOP has eventually been applied to a vast variety of quantitative areas, from finance to nuclear science.

Another noteworthy application of evolutionary computation can be found in Friedman (1959), whose theoretical work involved a robot progressively learning to move inside an arena. He also remarked how genetic algorithms could allow to develop thinking machines, somehow recalling Turing's early work.

A turning point in this field is represented by Fogel *et al.* (1966), which introduced the *evolutionary programming* technique. As the title of the book suggests, such approach was first used in order to generate an artificial intelligence. Differently from genetic programming, in this case the model subject to optimization cannot vary, while what is free to change and adapt its numerical parameters. Fogel's main idea was that one of the key features characterizing intelligence is the ability to predict: that's why its work is focused on using finite state machines [1] in predictive models, evolving their logics by means of selection and mutation.

Other relevant works from the period are Bledsoe (1961), Fraser and Burnell (1970) and Crosby (1973).

A new area of evolutionary field developed in the 1960s and consolidated in the 1970s: *evolutionary strategies*. First formalizations can be found in Rechenberg (1965) and Rechenberg (1973), who applied to equipments such as airfoils this parameters optimization method. Although being innovative, early Rechemberg's work had a major drawback: the population involved in the process was formed by only two individuals. The first one was the parent that, being mutated, gave birth to an offspring. Population size in evolutionary strategies did increase, but only in a second time. Further developments came from Schwefel (1975) and Schwefel (1977).

Among all the possible sources available, the greatest and most decisive contribution to *genetic algorithms* theory can be found in Holland (1975). His studies, started in the 1960s and continued throughout the 1970s at the University of Michigan, were originally not directly focused on the implementation of a problem solving mechanism, but were rather oriented towards a more general goal: studying natural adaptation and evolutionary phenomena in order to find a way to efficiently transpose them into computer science. The framework presented in the book *Adaptation in Natural and Artificial Systems* is so significant that it is still at the base of the modern genetic algorithm theory and its evolution.

[1] Programs that can be in a finite number of states, one state at a time. The machine starts in a state, receives inputs and generates transitions to one of the other possible states. Think, for example, to a game with a finite range of possible moves.

In Holland (1992) it can be read:

> Living organisms are consummate problem solvers. They exhibit a versatility that puts the best computer programs to shame. This observation is especially galling for computer scientists, who may spend months or years of intellectual effort on an algorithm, whereas organisms come by their abilities through the apparently undirected mechanism of evolution and natural selection.
>
> Pragmatic researchers see evolution's remarkable power as something to be emulated rather than envied. Natural selection eliminates one of the greatest hurdles in software design: specifying in advance all the features of a problem and the actions a program should take to deal with them. By harnessing the mechanisms of evolution, researchers may be able to "breed" programs that solve problems even when no person can fully understand their structure. Indeed, these so-called genetic algorithms have already demonstrated the ability to made breakthroughs in the design of such complex systems as jet engines.

Holland's genetic algorithm evolves a population of bit strings (*chromosomes* containing *genes*) by means of the formalization of natural processes like natural selection, mutation and recombination.

One of the major innovations was the possibility of exploiting a wide population together with all the basic operations (crossover, mutation, inversion), as opposed to aforementioned works such as evolution strategies and evolutionary programming, which adopted a limited amount of individuals and operations respectively.

A second great innovation introduced by Holland's work is the so-called *Holland's Schema Theorem*, which delivers a method to predict the goodness of the next generation in terms of above-average fitness individuals. A *schema* is a template describing a string of bits by means of two elements: fixed positions and wild-cards. The former are values which cannot vary, while the latter can. From this, it follows that the *order* of a schema represents the number of the fixed positions contained in the template and the *defining length*, instead, represents the distance between the first and the last fixed position. To make this clearer, consider the schema `01**1*0`: each `*` is a wild-card while the other values are fixed; the order is 4 and the defining length is 6. Lastly, the *fitness* of a schema is computed as the average fitness of all the strings that match the schema. Based on genetic algorithms methods, the theorem states that, as generations progress, schemata having an above-average fitness, a low order and a low defining length will increase exponentially.

The general interest around genetic algorithms kept growing until, in 1985, the *First International Conference on Genetic Algorithms* took place in Pittsburgh, Pennsylvania.

A few years later, the first non-academical product based on genetic algorithms, a mainframe toolkit for industrial purposes, was launched by General Electric while, in 1898, Axcelis, Inc. released *Evolver*, the first commercial product for desktop computers.

Eventually, a last branch of evolutionary computation evolved from genetic algorithms, and its formalization can be found in Koza (1992): *genetic programming*. The goal here is to avoid writing Lisp programs and, instead, automatically evolve them based on genetic algorithms.

1.2 The building blocks of evolving individuals

To start off with technical analysis, it can be useful to clarify some general biological concepts that, as already introduced in the previous paragraph while talking about Holland's work, are consistently applied to genetic algorithms. Although, notice how there is no perfect correspondence between biological and computer science terminology, since when talking about living entities things are much more complex. Such terms are used here in an intuitive way, with nothing but a similarity purpose.

Living organisms are made up of cells, which can be here thought of as containers for genetic information that defines all the creature's traits. Such information is encoded in *genes*, the fundamental units of inheritance, via DNA, which is the resultant of the mixing of individual molecules (the *nucleotides*). A usual but powerful way to understand this mechanism is to think about nucleotides as letters: once put together, they form words and sentences (DNA), creating instructions to build up the organism. When those DNA sentences are combined to create more complex instructions, they give birth to genes. Genes are located on and carried by *chromosomes*.

Genes can take a finite number of different forms, which will determine the corresponding organism's trait (think about hair or eyes color). Each of those genes versions is called *allele*. This is a key concept, since is only when alleles mutate that genes have the possibility to take different forms, thus to modify that specific creature's feature and carry on evolution.

So, DNA contained in genes can be encoded in a finite range of ways, which is defined as *genotype*. This is opposed to the *phenotype*, that represents only the set of the visible traits of a living being. To clarify this, consider a person inheriting from its parents both brown (B) and blue (b) eyes genes. The former are said to be *dominant* while the latter are said *recessive*, meaning that when mixing up together, brown and blue genes will result in brown eyes. In this case, even though the visible trait (the pheno-

type) is the brown one only, the chromosome still contains a Bb genotype, with both brow and blue alleles.

In evolutionary computation, as mentioned above, all those concepts and mechanisms are simplified, still maintaining a good correspondence with biology.

Given that the purpose of genetic algorithms is that of finding the best possible solution to a numerical problem, the chromosomes are represented by candidate solutions, typically under the form of strings or matrices of values. Such forms are not mandatory, but are largely widespread since they facilitate genetic operations such as crossover.

The genes, instead, correspond to appropriate numbers fitting the problem under consideration (can be binary values as well as integer or real numbers). What defines the proper range of values for the genes are alleles. Typically, two cases of alleles application can show up: one in which every gene contained in the chromosome should stay in the same range of values, thus creating the need to specify it only once; another one in which one or more genes must lie in their own specific ranges, making necessary to explicit different alleles for each one of them. This turns out to be very powerful in a wide variety of situations. Think about a simple portfolio optimization: my candidate solution may be a chromosome containing informations about quantities of stocks as well as stock prices: allowing the quantity genes to take the same values of the price genes would be not efficient, or even wrong. Defining a set of alleles equal to the market stock prices and another set corresponding to the possible quantities that can be bought (for example, taking into account the amount of products available on the market together with the buyer's budget) can make the operations much more efficient.

1.3 Encoding

All the above-mentioned elements, in order to be plugged into genetic algorithms, need to be formalized in computer language, and such process requires a careful analysis of the specific problem being faced. Depending on the given set up, indeed, chromosomes should contain the proper kind of genes, thus to avoid further bugs and conflicts while running the algorithm.

In order so make this more clear, some of the most common encoding cases are presented:

Binary encoding: is probably the simplest and most widespread case. For instance, almost all the early works in the genetic algorithms field featured such encoding. Working with this kind of encoding means defining the chromosomes as bit strings, whose genes can only assume 0 or 1 value. Such set up is simple, maintaining a low number of alleles while still allow-

ing non-trivial individuals to be generated. The main drawback, however, is that for most applications a binary representation will not fit, requiring a wider range of possibilities and characteristics.

Here is an example a of binary encoded chromosome:

```
0011001011101001
```

Value encoding: allows a more complex representation to be implemented. This is crucial when trying to solve problems involving elements which can be in more than two different states.

Working with financial markets, for instance, means handling prices, budgets and quantities. While it is possible to exploit binary encoding to identify the fact that an agent is or is not participating to the negotiations, the previous features evidently require a different approach.

Value encoding allows any kind of genes representation, be it letters or real numbers, or even other elements such as a series of commands. Here are some examples:

Example A: `T C A G C C T G A C A A G`
Example B: `12.56 5.34 99.0 54.01 23.78`
Example C: `(up)(left)(up)(right)(right)(down)`

Permutation encoding: this is a more specific kind of encoding, only useful when considering ordering problems. One of the most famous examples of this is the *travelling salesman problem*: having a list of cities the salesman needs to visit, and given the distances between each pair of cities, which is the most efficient (i.e. the shortest) possible route which allows to come back to the starting point after having visited each city once? Without diving any deeper in the many implications of this problem, just consider the kind of encoding it requires. Here is an example:

```
659438217
```

where each value denotes a city and their ordering defines a possible route for the salesman.

1.4 Fitness function

The starting point of every genetic algorithm is the problem to be solved. Whether it is a maximization, a minimization or an equation solving problem, the algorithm is always asked to perform an optimization operation which can be formally represented by means of a mathematical function. The population of candidate solutions, then, is plugged into such function in order to check their strength and to consequently perform the adequate

evolutionary operations. This level of goodness is named *fitness* and, hence, the initial function used to test it is named *fitness function*.

Consider, for instance, the *coin collector's problem*: imagine a coin collector having a collection of n coins, each one with a different numismatic value. Suppose the collector needs to buy something having cost c, but has run out of money. The only solution is to sell some of his coins, but of course the collector wishes to give away the lowest possible amount of coins. Formally:

initial set of coins: $A = \{a_1, a_2, ..., a_n\}$
subset of coins to be sold: $A \supseteq B = \{b_1, b_2, ..., b_m\}$
collector's goal: $\sum_{j=1}^{m} b_j = c$

Adopting a binary representation, a bit string denotes the subset of coins to be sold (B), each bit denoting a single coin (b_j). Bits will obtain value 0 if they don't fit the problem (i.e. $b_j > c$), 1 otherwise ($b_j \leq c$). Fitness will correspond to the sum of 1s contained in the subset satisfying the collector's goal, representing the number of candidate coins to be sold.

Notice how, in this particular case, the problem requires a fitness minimization, showing how constructing a genetic algorithm and a related fitness function is not a mechanical, uncritical process but needs adaptation to the particular case.

1.5 Selection methods

After a fitness function has been defined, as well as the initial population (or *generation 0*), the genetic algorithm is required to somehow mimic the biological process of natural selection. Hence, based on some arbitrary principle, the individuals that better fit the initial problem will be rewarded with the possibility to breed a new generation, bequeathing their traits to the newborn individuals; individuals performing a poor fitness rate will instead exit the evolutionary process.

Depending on the principle adopted to select the reproducing individuals, different selection methods take place:

Fitness proportionate selection (or **roulette wheel selection**): as for any selection method, the first step here is to utilize the fitness function to evaluate and assign to each candidate solution a fitness value (f_i). Supposing the total number of individuals contained in the population is N, the total fitness is computed by means of a simple summation: $\sum_{i=1}^{N} f_i$.

At this point, it is possible to compute the probability of each individual to be selected (p_i) by just normalizing the fitness values, as follows:

$$p_i = \frac{f_i}{\sum_{i=1}^{N} f_i}. \tag{1.1}$$

Once this is done, the candidate solutions are sorted by decreasing fitness value so to apply a cumulative probability distribution (CDF) computation, meaning the second normalized fitness value is summed to the first, then the third value is summed to the previous two, and so on until the last value, eventually summing up to 1.

To select an individual, a uniform random number U from the range $[0, 1)$ is generated and the inverse of the CDF is computed in such value. In other words, the solution that is going to be selected is the one that, when added to the cumulative computation, returns a value greater than U.

The whole process may resemble a roulette wheel spin: the total fitness is normalized so to sum up to 1 (just like a roulette wheel does) and each slice represents an individual. This is actually a sort of enhanced roulette wheel, since fitter solutions obtain greater slices. The uniform random number and inverse CDF technique, then, simulates a wheel spin, which must be executed as many times as many individuals are desired to be selected.

Notice how adopting a fitness proportionate selection does not guarantee in any way that only the best individuals will be picked: while it is true that the greater the fitness, the greater the chance to be chosen, all the individuals have are equipped with a non-zero probability. This is not necessarily a disadvantage, since weaker solutions may posses some useful traits that cannot be found in the strongest ones, hence promoting variety and diversification.

Stochastic universal sampling (SUS): first introduced in Baker (1987), this method is a development of the fitness proportionate selection, aimed at overcoming a specific drawback know as *premature convergence*. Roulette wheel, indeed, is exposed to the risk that one single individual, which can turn out to be by far the fittest in comparison to the rest of the population, can heavily bias the overall performance of the selection. In such situation, the risk is that too many inverted CDF computations give, as a result, the largely fit solution, excluding all the other ones. In other terms, stochastic universal sampling ensures the minimal spread between the expected and the observed selection frequencies. Consider, for instance, a candidate solution i whose probability of being selected is $p_i = 7.5\%$: picking 100 individuals, i is expected to show up between 7 and 8 times. This is what, differently from roulette wheel selection, is ensured by SUS.

More specifically, stochastic universal sampling can still be imagined as

dividing a wheel proportionally to the normalized fitnesses, as computed in 1.1. This time, however, one unique wheel spin is needed: a uniform random number in [0, 1) is generated, and this represents the starting point of the selection. From now on, no spins are needed because subsequent individuals are picked at fixed constant intervals. Namely, wanting to select N solutions, the intervals length equates $1/N$.

Clearly, as happens with fitness proportional selection, SUS does not prevent weaker individuals from being selected.

Truncation selection: this is one of the less sophisticated selection methods and, differently from what has just been explained for roulette wheel selection and stochastic universal sampling, it allows to permanently eliminate the weakest candidates from the evolutionary process.

Again, the first step is to use the fitness function to rate the goodness of each solution and, consequently, order them based on their strength. What decides which individuals will survive is the *truncation threshold*: this value simply indicates the proportion of the population that must breed and produce offspring. Last step is about replacing the candidates which did not pass the threshold, and this is done by replicating the fittest individuals a suitable number of time.

To clarify: if the threshold t equals $1/2$, then after the truncation the remaining candidates will be doubled to restore the initial population size. In general, for $t = 1/x$, the surviving individuals are reproduced $1/t$ times.

Tournament selection: is widely used and offers several advantages. This method is implemented by running a series of tournaments (as many as the number of individuals desired) among the population.

A tournament consist in randomly picking 2 or more candidate solutions and comparing their goodness, thus to select for breeding the fittest individual taking part to the tournament. This way of operating easily allows to control for the selection pressure[2]: allowing tournaments to involve more individuals increases the competition, resulting in a lower probability of survival for weaker candidates. The method allows to remove the individual once it has been selected or to keep it competing. Of course, reducing the tournament size to just 1 candidate results in nothing but random selection. Another advantage with tournament selection is the fact that works well with parallelization.

Sigma scaling: speaking again about premature convergence, its cause can be seen from a different point of view. The presence of one or more highly fit individuals translates into a great value of the population variance. Such bias leads, at early evolutionary stages, to a massive reproduction of a few

[2]The degree to which highly fit individuals are allowed many offspring.

strings, to the detriment of the rest of the population. On later stages, on the contrary, the candidates will typically have the same level of goodness (i.e. a low variance), thus to almost halt the whole process. So, in conclusion, it is evident how premature convergence is a variance-driven phenomenon.

To contrast this, a solution may be *scaling*. Sigma scaling, as introduced in Forrest (1985) and Goldberg (1987) (here named *sigma truncation*), keeps under control the selection pressure by computing individual's expected values as a function of their fitness as well as of the population's mean and standard deviation. As an example, call $f(i)$ the individual's fitness, $\overline{f}(t)$ the population fitness mean at time t and $\sigma(t)$ the fitness standard deviation at time t. Here is how the expected value function is computed in Tanese (1989):

$$ExpVal(i,t) = \begin{cases} 1 + \dfrac{f(i) - \overline{f}(t)}{2\sigma(t)}, & \text{if } \sigma(t) \neq 0 \\ 1.0, & \text{if } \sigma(t) = 0 \end{cases} \tag{1.2}$$

This allows, in the first generations, for candidate solutions with high variances to have an expected value not too distant from the mean value, while giving a low opportunity of being selected to the weakest individuals as well. At the later stages of the evolution, this same mechanism fights the possible stagnation, giving to fitter strings the possibility to reproduce.

Rank selection: this method is another way to contrast premature convergence. In this case, indeed, slices of the wheel are not assigned based on absolute probabilities of being selected, but rather based on relative probabilities. First thing the method does, in fact, is to rank each individual based on its p_i, computed according to 1.1. Once this is done, the chromosomes obtain a portion of the wheel whose dimension is determined accordingly to the individual's rank. Fitter individuals are associated to larger slices. It is now clear how extremely large-fitness individuals do not have the possibility of biasing the selection, since the resulting roulette wheel is always subdivided in the same way.

Various ranking methods can be applied, either linear or non-linear ones. The advantage of choosing the latter over the former is that of obtaining a wider range of values for selecting pressure: from [1, 2] to [1, N-2], where N denotes the population size.

As an example, the linear ranking proposed in Baker (1985) ranks the N individuals increasingly based on their fitness. The user is then asked to declare the expected value ($Max \geq 0$) associated with the individual with rank N. Calling Min the expected value of the solution ranked 1, the expected value of each individual in the population, at time t, is given by:

$$ExpVal(i,t) = Min + (Max - Min)\frac{rank(i,t) - 1}{N - 1}. \tag{1.3}$$

It will follow, as said, that $1 \leq Max \leq 2$ and $Min = 2 - Max$.

Elitism: this is not a proper selection method but, rather, a way to enhance a method.

Sometimes, indeed, highly fit individuals can be lost due to crossover and mutations, even resulting in weaker candidates. Elitism, firstly introduced in De Jong (1975), ensures this will not happen by copying a portion of the fittest individuals of the population, plugging them directly into the following generation without any kind of modification. This process, however, does not prevent the same individuals to be also chosen as parents for breeding.

This way of proceeding turns out to have a significant positive impact on genetic algorithm operations, notably improving the quality of the individuals involved.

1.6 Genetic operators

Once the proper selection method has been applied to the population, the next step consists in giving birth to a second generation which is required to encapsulate all the good features that made possible for their parents to survive and bequeath their genes. This can be achieved by means of the so called genetic operators: *crossover* (or *recombination*) and *mutation* are the main ones and by far the most utilized, but other approaches such as *migration*, *regrouping* or *colonization-extinction* are available.

To create new chromosomes, the program picks one or more individuals from the previous generation, among the ones that passed the selection barrier, and applies genetic operators to them, in order to manipulate their features and transmit them to the newborn candidates. This process is iterated as many times as needed to reach the desired population size.

The typical choice is to select two parents to run such process, and this also means to keep the algorithm closer to the biological events. However, it is possible to utilize more than two individuals at a time and this seems to give the chance for a higher quality breeding, as stated in Eiben *et al.* (1994) and Ting (2005).

Commonly, this whole procedure results in an increased average population fitness, not only thanks to the genetic operations but also to the previous selection work, which ensure the dominance of the most fit individuals together with the preservation of genetic diversity.

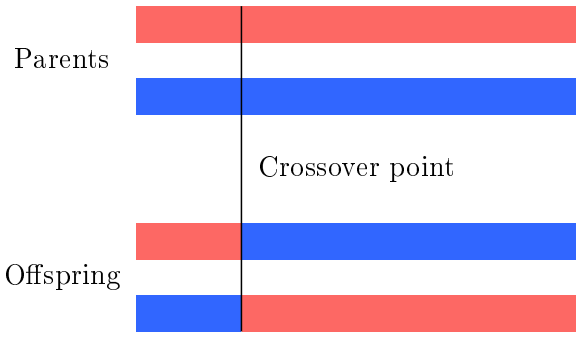
As already said, the main operators are crossover and mutation, so they are presented in the following sections.

Crossover

This process is based on the same principles that can be found in biological reproduction and crossover. Generally speaking, the idea is to choose two or more individuals which will be manipulated and mixed in order to give birth to a child solution. Those are the main techniques to be adopted:

Single-point crossover: one unique crossover point is randomly selected and applied to both parents. All the information encoded beyond that point is swapped in order to generate two new chromosomes.

Graphically:

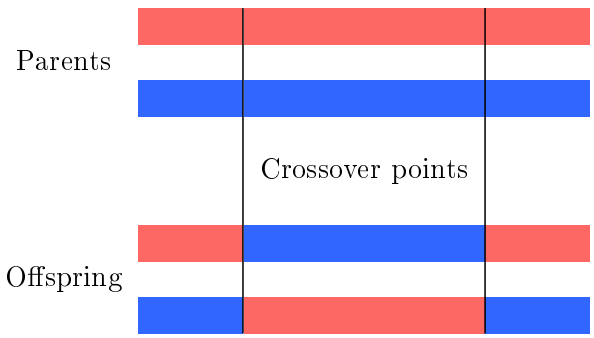


Numerically:

ParentA:	**1**	**0**	**0**	1	1	0	1	1	0	1
ParentB:	0	1	0	**0**	**1**	**0**	**0**	**1**	**1**	**0**
ChildA:	**1**	**0**	**0**	**0**	**1**	**0**	**0**	**1**	**1**	**0**
ChildB:	0	1	0	1	1	0	1	1	0	1

Two-point crossover: two different crossover points are randomly selected and applied to both parents. Any information contained between those points is swapped, giving birth to new children.

Graphically:

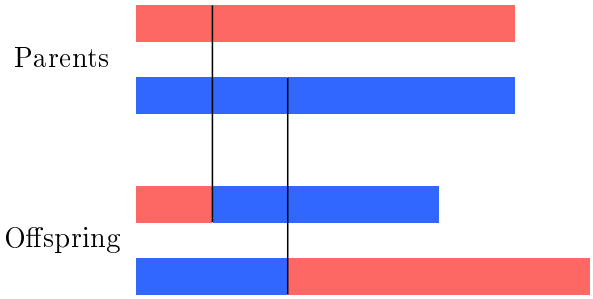


Numerically:

ParentA:	**1**	**0**	**0**	1	1	0	1	**1**	**0**	**1**
ParentB:	0	1	0	**0**	**1**	**0**	**0**	1	1	0
ChildA:	**1**	**0**	**0**	**0**	**1**	**0**	**0**	**1**	**0**	**1**
ChildB:	0	1	0	1	1	0	1	1	1	0

Cut and splice: is a variation that typically leads to a change in the chromosomes length. This is due to the fact that one or more random crossover points are independently selected for each one of the parents.

Graphically:

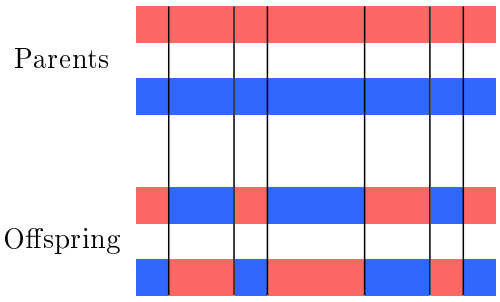


Numerically:

ParentA:	**1**	**0**	**0**	1	1	0	1	1	0	1
ParentB:	0	1	0	0	1	0	**0**	**1**	**1**	**0**
ChildA:	**1**	**0**	**0**	**0**	**1**	**1**	**0**			
ChildB:	0	1	0	0	1	0	1	1	0	1	1	0	1

Uniform and half uniform crossover: this technique exploits fixed mixing ratios rather than fixed crossover points. Choosing a mixing ration equal to 0.6, for example, the newborn individuals will feature 60% of the first parent's genes and 40% of the second one's. Crossover points are still used, but they are randomly chosen and serve the only purpose of defining areas for which genes will be evaluated for exchange.

Graphically:

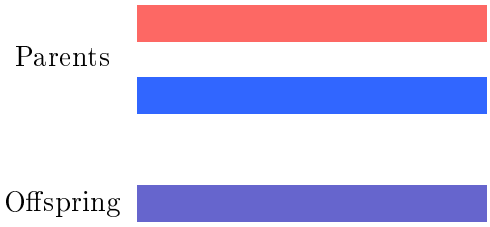


Numerically:

ParentA:	**1**	0	**0**	**1**	1	0	**1**	1	**0**	1
ParentB:	0	**1**	0	0	**1**	**0**	0	**1**	1	**0**
ChildA:	**1**	**1**	**0**	**1**	**1**	**0**	**1**	**1**	**0**	**0**
ChildB:	0	0	0	0	1	0	0	1	1	1

Arithmetic crossover: two parents are linearly combined according to a predefined equations system.

Graphically:



Numerically: given a weighting factor $a = 0.7$ and an equations system corresponding to

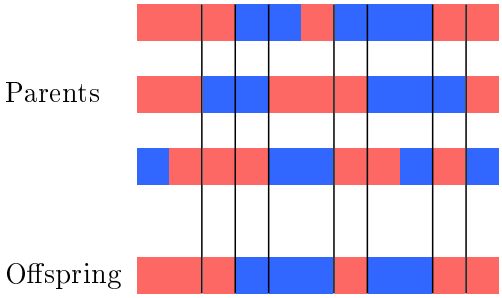
$$ChildA = a * ParentA + (1 - a) * ParentB$$
$$ChildB = (1 - a) * ParentA + a * ParentB$$

the resulting crossover will be

ParentA:	(0.3)	(1.4)	(0.2)	(7.4)
ParentB:	(0.5)	(4.5)	(0.1)	(5.6)
ChildA:	(0.36)	(2.33)	(0.17)	(6.86)
ChildB:	(0.402)	(2.981)	(0.149)	(6.842)

Three-parent crossover: this method generates one single child from the information contained in three different individuals. The technique consists in comparing genes from the first two parents: if they are the same then such gene will be transmitted to the child, otherwise the third parent's genes is inherited.

Graphically:



Numerically:

ParentA:	1	**1**	0	1	**0**	0	**0**	1	0
ParentB:	0	**1**	1	0	**0**	1	**0**	0	1
ParentC:	**1**	1	**0**	**1**	1	**0**	1	**0**	**1**
Child:	**1**	**1**	**0**	**1**	**0**	**0**	**0**	**0**	**1**

Permutation encoding crossover: when chromosomes represent specific orderings single-point crossover can be applied, with some variations with respect to the binary and value encoding cases. The first parent, indeed, is still split as usual but, to complete the resulting offspring, the second parent is scanned: if a gene is not yet present in the child (coming from the first parent) it is added.

Numerically:

ParentA:	**1**	**3**	**5**	**4**	**2**	9	6	8	7
ParentB:	4	**9**	1	2	**7**	5	**6**	**8**	3
Child:	**1**	**3**	**5**	**4**	**2**	**9**	**7**	**6**	**8**

Mutation

Inspired by biological mutation, this process is aimed at ensuring genetic diversity throughout the generations of candidate solutions. It operates by picking one or more genes from the chromosomes and changing their value in accordance to the selected principle. Just like their biological counterparts, individuals can obtain a benefit as well as a disadvantage, since mutation (and crossover too) is an impartial mechanism.

During the evolutionary process, mutations happen at a user-defined rate. Typically, such value is set to be low: otherwise, risks are those of premature convergence or of turning the algorithm into a mere random search. A moderate mutation, indeed, boosts evolution by avoiding individuals to become too much similar to each other, situation which is the prelude to a stop in the process.

Those are the most used kinds of mutation:

Bit string mutation: such method applies only to binary encoding. One or more genes are picked at random from the chromosome and are then flipped.

Numerically:

Before mutation:	0	1	**0**	0	1	0	**1**	1
After mutation:	0	1	**1**	0	1	0	**0**	1

The probability of mutation for each gene is $1/l$, l denoting the chromosome's length.

Gaussian mutation: applicable only to real valued genes, this technique adds a standard normally distributed random number to one or more elements of the chromosome.

Numerically:

Before mutation:	(0.3)	(1.4)	(0.2)	**(7.4)**
After mutation:	(0.3)	(1.4)	(0.2)	**(7.4+σ*N(0, 1)*)**

Uniform mutation: can be used with real valued genes only. It picks one genes and replaces it with a random number generated from a user-defined range.

Numerically:

Before mutation:	(0.3)	(1.4)	(0.2)	**(7.4)**
After mutation:	(0.3)	(1.4)	(0.2)	**(*U(a, b)*)**

with a and b being the arbitrary boundaries.

Boundary mutation: can be used only with real valued genes. The selected element is randomly substituted with either the upper or lower bound (selected by the user).

Numerically, with a and b being the boundaries:

Before mutation:	(0.3)	(1.4)	(0.2)	**(7.4)**
After mutation:	(0.3)	(1.4)	(0.2)	**a**

Order changing: typically used in permutation encoding cases, this kind of mutation selects two random positions and swaps their values.

Numerically:

Before mutation:	2	6	**5**	8	4	**1**	9	3	7
After mutation:	2	6	**1**	8	4	**5**	9	3	7

1.7 Termination

After having defined the problem to be solved, chromosomes are properly encoded and a suitable fitness function is defined. An initial population is generated and then evaluated applying a selection method. Thanks to this process, the best individuals obtain the right to bequeath their genes to the following generation, and this is done by means of crossover and mutation operators. A second generation is born and the cycle is iterated.

This, in short, is the whole general structure of any genetic algorithm. One last problem is left to be faced: when to stop the evolutionary operations. This is not a trivial issue, since *termination criteria* can turn out to have a significant impact on the final performance of the whole program. On one hand, indeed, they are a useful tool in order to prevent needless computations from being executed: at some point, it can happen that the efficiency of the algorithm is exhausted, so that the operations degenerate to a mere random search and no improvement is to be expected in terms of candidate solutions. In such situation, the time and energy waste is evident. On the other hand, a bad handling of the criteria can result in a premature termination, stopping the algorithm before a satisfying solution is figured out.

Bearing these aspects in mind, the criterion construction possibilities are virtually limitless. Here, some of the most common techniques are presented.

Elapsed time: the user sets a specific amount of time which, once passed by, forces the program to end.

Stall time limit: is a variant of the previous approach. In this case, the process is run aiming at improving the goodness of the candidate solutions. However, if during an arbitrary interval of time (the *stall time limit*) no fitness improving is observed, the iteration is ceased.

Maximum generations: regardless of the time elapsed, the algorithm evolves a predefined number of generations and then stops.

Stall generations: similarly to the stall time limit case, the process stops if no fitness improvement is observed for a predefined amount of consecutive generations, called *stall generations*.

Minimum fitness: to stop the algorithm, in this case it suffices that one single individual reaches a user-defined minimum goodness level.

Problem solved: this method is applicable only in some specific cases. Namely, only when the problem being faced admits one single solution. Typically, the fitness function is constructed such that, when and only when such

solution is found, it takes value 1 (or any other predetermined value). This situation is the signal that makes the algorithm halt.

Statistical criteria: this family of techniques relies on the computation and comparison of some statistical features of the current generation. For instance, the process may stop when the difference between the fitness of the best individual and the population's mean fitness passes a threshold ε, or when the standard deviation is less than or equal to ε, or when the difference between the best and the worst objective value exceeds ε.

Chapter 2

Portfolio theory

When speaking about the theory behind current portfolio selection techniques, it is inevitable to start off by describing the element that represents the biggest cornerstone in this field: *Modern Portfolio Theory*, also known as *mean-variance analysis* or *mean variance optimization*. The first, fundamental step has been made with Harry Markowitz (1952), article in which the idea was firstly presented. Almost four decades later, in 1990, together with Merton Miller[1] and William Sharpe[2], Markowitz was awarded the Nobel prize for his theory.

In its basic version, the Modern Pricing Theory gives the investor the foundations to construct an optimal investment portfolio, taking into consideration two main elements: the foreseen assets return and the investor's aversion to the risk they bear.

Before analyzing in further details the mean-variance optimization, it may be useful to clarify some preliminary aspects which can be found at the very base of portfolio theory.

2.1 Risk propensity

Observing the historical market data on various asset classes it is always possible to spot a common characteristic, no matter which type of investment nor which time period is under consideration: higher levels of risk associated to the assets correspond to higher level of guaranteed returns. This simple, maybe intuitive, information leads to a fundamental concept: in order to expose themselves to any sort of risk, investors want to be remunerated accordingly.

[1] For his work in corporate finance field.

[2] For his contribution to *Capital Asset Pricing Theory* and, in general, to price formation theory.

From here, the concept can be expanded and, more generally, it can be said that all entities (individuals as well as firms or funds) participating to the market show specific propensities to risk. All investors attribute a greater preference to those assets promising a higher return, and this represents the common starting point in the evaluation of an investment. What makes the difference among them, however, is how such initial level of preference is modified in accordance to the risk the asset is associated to. There are three possible outcomes:

Risk aversion: this approach leads the investor to penalize any possible level of risk. Wanting to vaguely formalize it, this translates into a negative coefficient associated to risk:

$$preference = expected\ return - \lambda(risk).$$

As mentioned above, this is by far the most widespread kind of approach to risk, the one responsible for the *higher risk = higher return* situation observed in markets.

Risk neutrality: a neutral entity is completely indifferent to the level of risk being faced. This translates into $\lambda = 0$ which, by consequence, means that the preference is solely determined by the promised return, without any sort of penalization:

$$preference = expected\ return.$$

Risk propensity: investors falling into this category are risk lovers who add even more value to the expected return when it comes with higher levels of risk. In such case the observed λ is positive:

$$preference = expected\ return + \lambda(risk).$$

Such entities, for example, always agree to take a fair game[3] instead of choosing a risk-free investment.

This trade-off between return and risk can be intuitively represented as in the simple example provided in Bodie *et al.* (2013). It is sufficient to plot investors choices in a graph having the expected portfolio returns, $E(r)$, on the vertical axes and the portfolio risk, σ, on the horizontal axes:

[3] A risky investment showing a zero risk premium.

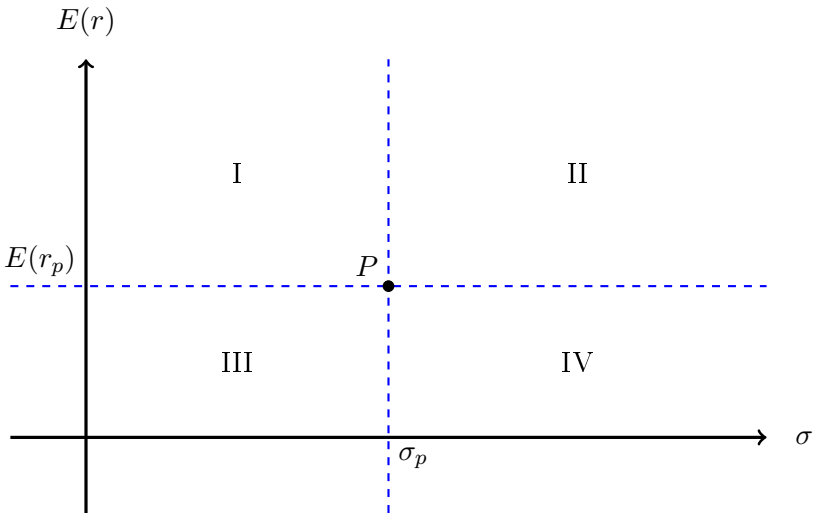


Figure 2.1: Trade-off between risk and return

Point P represents one specific portfolio possibility, featuring corresponding levels of expected return ($E(R_p)$) and risk (σ_p). Taking it as a starting point, it is possible to analyze an investor's preferences as a response to a change in one of the two variables involved in the trade-off.

Considering a risk-averse individual, it is out of doubt that any portfolio located in quadrant I would be preferred to P, since it would offer a higher return at a lower risk level. Conversely, any asset combination falling into quadrant IV would be rejected, exposing the investor to a higher risk while offering lower expected returns.

Such approach is named *mean-variance criterion*, and it can be generalized saying that the conditions under which portfolio A is always preferred to portfolio B (in a risk aversion case) are:

$$E(r_A) \geq E(r_B)$$

and

$$\sigma_A \leq \sigma_B$$

with at least one strict inequality. Two equalities, of course, would make the investor indifferent to portfolios A and B.

So far, the analysis has been trivial and gave birth to no doubt. However, half of the graph is still unexplored: what happens to an individual's preferences when she moves to quadrants II and III? The answer cannot be unique here, since return and risk move in the same direction. To know exactly what the outcome would be, a key element is missing: a specification of investor's risk aversion.

2.2 The utility function

Suppose the investor from Figure 2.1 plots on the mean-variance plane all the portfolios which, to her, are as attractive as portfolio P. All such points, since she is risk-averse, will lie in quadrants II and III. Indeed, an increase in risk (σ) must be compensated with the promise of a higher remuneration ($E(r)$) while a lower expected return can be accepted only in the presence of a lower possibility of loss:

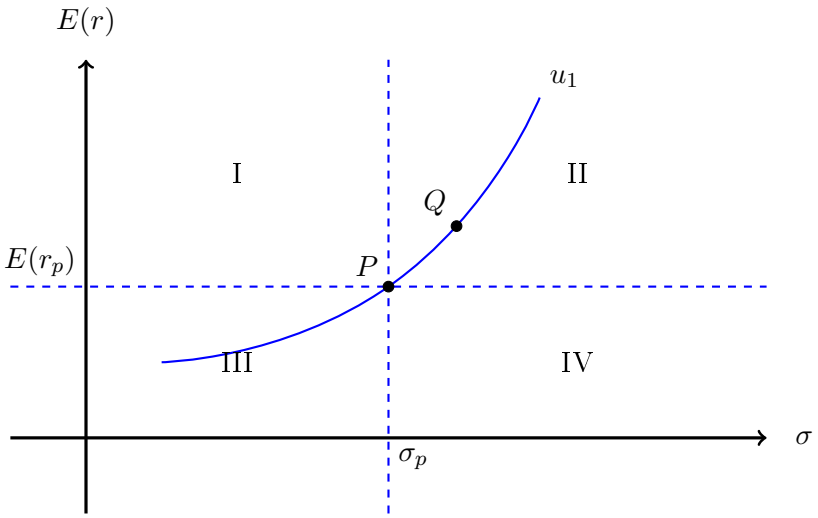


Figure 2.2: The indifference curve

It results that, in Figure 2.2, portfolio Q is as desirable as portfolio P, and this is also true for any other point lying on the curve u_1. Given that any point chosen from u_1 gives the same level of satisfaction to the investor, such line is called *indifference curve*.

Figure 2.2, however, is still limited: it represents only one level of utility, taking into account only a restricted set of points. What happens if the investor picks a portfolio lying above or below u_1? Intuitively, keeping fixed the level of risk (say, σ_p), to move upwards means obtaining a higher return, while moving downwards results in lower remuneration. Hence, based on the mean-variance criterion stated in section 2.1, this translates into greater and smaller preference respectively. Again, for such new risk-return combinations it is possible to plot the corresponding indifference curves:

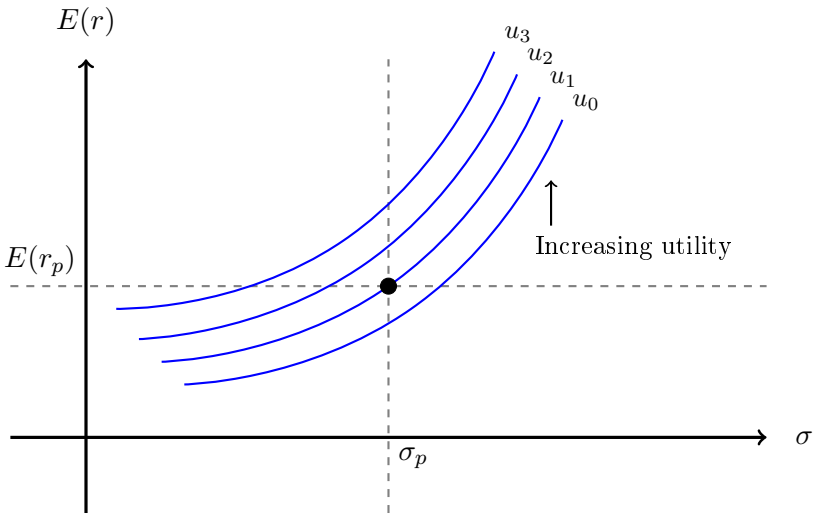


Figure 2.3: A set of indifference curves

As said with regards to Figure 2.2, in Figure 2.3 any point lying on the same curve represents a portfolio giving to the investor the same level of satisfaction. The relationship between the curves, then, is simple: the utility given by portfolios belonging to u_3 is greater than that given by u_2 portfolios which, at the same time, are more satisfactory than portfolios forming u_1. Lastly, u_0 ensures the lower utility.

The representation from Figure 2.3 is a graphical way to picture a so-called *utility function*, which is an instrument coming from the economic choice theory used to describe the behavior of entities facing a choice between two or more options. The utility function serves the scope of translating into numbers (called *utility indexes*) such possible choices, following a simple scheme: as long as the utility provided by option a is greater than the one obtainable from option b, then a is preferred to b.

In the case here considered, utility functions must satisfy the following analytical requirement: they have to be twice continuously differentiable such that $u' > 0$ and $u'' \leq 0$. This means that an investor always prefers more to less utility but also that marginal utility is decreasing in wealth.

Having said this, do not forget that the goal here is to specify investors risk aversion, and utility functions can do it well. In order to do so, however, a numerical representation of the function is needed. Once the specification is known, it is then possible to compute two measures of an individual's aversion to risk: *relative* and *absolute risk aversion*. In general, they are given by:

$$r_R = -x\frac{u''(x)}{u'(x)}$$

and

$$r_A = -\frac{u''(x)}{u'(x)}$$

respectively. As for the numerical representation, any form is admitted, but there exist a set of them which is commonly used:

Linear utility function:

$$u(x) = a + bx$$
$$r_A(x) = r_R(x) = 0$$

Zero absolute and relative risk aversion make a linear representation suitable for a risk-neutral investor.

Quadratic utility function:

$$u(x) = x - \frac{b}{2}x^2, \quad b > 0$$
$$r_A(x) = \frac{b}{1-bx}, \quad r_R(x) = \frac{bx}{1-bx}$$

Exponential utility function:

$$u(x) = -\frac{1}{\lambda}e^{-\lambda x}, \quad \lambda = 0$$
$$r_A(x) = \lambda, \quad r_R(x) = \lambda x$$

Given the nature of its absolute risk aversion, this function is often referred to as *constant absolute risk aversion* (CARA).

Power utility function:

$$u(x) = x^\alpha, \quad 0 < \alpha < 1$$
$$r_A(x) = \frac{1-\alpha}{x}, \quad r_R(x) = 1 - \alpha$$

Similarly to what happens for the exponential utility, the power utility function is commonly known as *constant relative risk aversion* (CRRA).

Logarithmic utility function:

$$u(x) = ln(x)$$
$$r_A(x) = \frac{1}{x}, \quad r_R(x) = 1$$

Relative risk aversion is constant again. This must not surprise, since the logarithmic representation consists in nothing but a specific power utility function case with $\alpha = 0$.

2.3 Diversification

Another aspect to be clarified is the role of diversification in the investment process. It may seem common sense to not employ all one's resources in buying one single asset, but there exists a stronger mathematical foundation to this. Markowitz adopted two forms of measure for quantifying assets risk: the *variance*, which tells how wide is the range of possible values for the expected return, and the *covariance* between assets returns, with which it is possible to capture the degree at which a fluctuation in one asset's return influences another asset's return. Ideally, an investor would want low variances, thus to avoid undesired return falls, and low covariances, thus to protect the portfolio from shocks that would affect all the assets. The way to reach this conditions is diversification.

At the base of diversification theory is the *Central Limit Theorem*, saying that the asymptotic sum of a series of identically and independently distributed random variables with bounded variances is Gaussian. Formally:

> Being $X_1, X_2, ..., X_N$ N identically and independently distributed random variables with finite mean μ and finite variance σ^2 then, for N going to infinity:

$$\lim_{N\to\infty} P\left(\frac{1}{\sigma\sqrt{N}}\sum_{i=1}^{N}(X_i-\mu)\leq y\right) = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{y} e^{-\frac{1}{2}s^2}\,ds$$

From here it follows that, for an equally weighted portfolio containing N assets with returns $R_1, R_2, ..., R_N$, the portfolio return will be:

$$R_p = \frac{1}{N}\sum_{i=1}^{N} R_i$$

which, for the Central Limit Theorem, is a Gaussian random variable when N is sufficiently large. This allows to compute the portfolio variance as follows:

$$var(R_p) = \frac{1}{N^2}\sum_{i=1}^{N} var(R_i)$$

$$= \frac{1}{N^2} N\sigma^2$$

$$= \frac{\sigma^2}{N} \xrightarrow{N\to\infty} 0.$$

In words: as the number of assets N grows to infinity, the portfolio variance decreases towards zero. Of course, this is an ideal situation not

achievable in reality. However, it is still a solid explanation of the impact that diversification has on a portfolio's risk.

Such set up is not always verified: some assets show fat tails in their return distributions, due the their non-normality; some assets have not bounded, hence infinite and so non-existent, variances; as shown in Fama (1965), stable Paretian distributions make diversification a meaningless economic activity. However, consensus on diversification is wide and the practice is widely applied.

Now that those concepts have been clarified, it is possible to move on to Markowitz's approach to portfolio selection.

2.4 The mean-variance approach

To the end of this analysis, the subject considered is a rational, risk-averse investor who operates at time t for the time limit $t + \Delta t$. No attention is given to what may happen before or after $t + \Delta t$, and new decisions are taken only after the first time period of length Δt is expired. This kind of approach is named *myopic* and is generally suboptimal, but is picked here for sake of simplicity. As already said, Markowitz's assumption is that the rational investor takes decisions based on a trade-off between expected return and risk. More specifically, *expected return* is defined as the expected asset price change plus any possible income obtained during the time period (for instance, dividends payment). This sum is then divided by the initial asset price. The *risk*, as already explained, is measured as the variance of returns.

In theory, there exist infinite possibilities of assets combinations in order to construct a portfolio, but not all of them can actually be created. The set of the possible portfolios is called *feasible set*. Among all the possible trade-off opportunities and for a given level of expected return, then, an investor would pick the portfolio with minimum variance, which takes the name of *mean-variance efficient portfolio*. Plotting on the mean-variance plane all the feasible mean-variance efficient portfolios gives birth to the so-called *efficient frontier*:

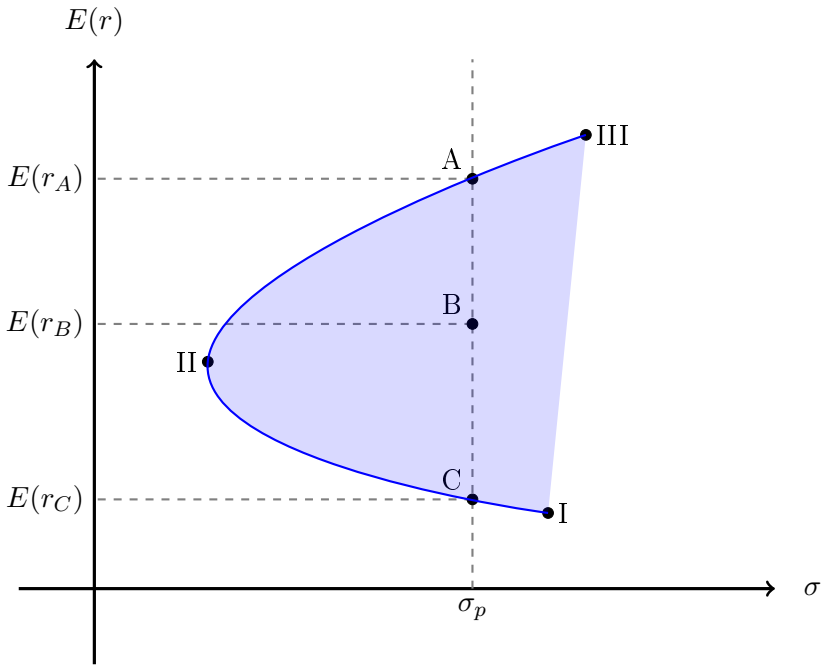


Figure 2.4: Feasible and efficient portfolios

The colored area in Figure 2.4 denotes the feasible set, while the curve passing through point I, II and III represents the efficient frontier. This can be seen considering point A: keeping fixed the expected return level at $E(r_A)$, any portfolio lying on its left would fall outside the feasible set (i.e. would be impossible for the investor to hold it) while moving to its right would result in a feasible but riskier portfolio.

The efficient frontier, then, can be further broke down into two different parts individuating the so-called *global minimum variance portfolio* (GMV), which in Figure 2.4 is represented by point II. The GMV is the portfolio belonging to the efficient frontier that features the lowest variance. Now, it is evident that the portion of curve going from II to III displays an even higher level of efficiency: as a piece of the efficient frontier, it still individuates the lower risk for a given level of expected return but, at the same time, it maximizes the remuneration for any given level of variance. Consider points A, B and C: they share the same σ value, but it is evident how A, lying on II-III, ensures by far the highest $E(r)$ value.

Please notice that, in the real process of portfolio construction, expected return, volatility and correlation values are not known exactly, but need to be estimated. However, for sake of simplicity, they will be assumed to be given.

N risky assets

Now, it is possible to begin analyzing the method with a more rigorous mathematical approach. First of all, imagine the investor has the possibility to build a portfolio containing N risky assets. This translates into N percentages of the total portfolio to be assigned to each asset, which take the name of *weights*. They can be collected into an N-vector $\mathbf{w} = (w_1, w_2, ..., w_N)'$[4], with the requirement that they sum up to 1:

$$\sum_{i=1}^{N} w_i = 1.$$

At the same time, each asset features a return such that $\mathbf{r} = (r_1, r_2, ..., r_N)'$ and such that the expected returns are denoted as $\boldsymbol{\mu} = (\mu_1, \mu_2, ..., \mu_N)'$. Lastly, the assets give birth to an $N \times N$ variance-covariance matrix:

$$\boldsymbol{\Sigma} = \begin{bmatrix} \sigma_{11} & ... & \sigma_{1N} \\ \vdots & \ddots & \vdots \\ \sigma_{N1} & ... & \sigma_{NN} \end{bmatrix}$$

Here, σ_{ij} is the covariance between assets i and j. It follows that $\sigma_{ii} = \sigma_i^2$ represents the variance for asset i. Moreover, exploiting the definition of correlation, it is possible to write: $\sigma_{ij} = \sigma_i \sigma_j \rho_{ij}$.

Notice that, for the time being, no constraints are applied and, in particular, short selling is allowed (i.e. w_i can take negative values).

With such notation, it results that the return of a N-risky asset portfolio, given by the weighted returns, is:

$$R_p = \mathbf{w}'\mathbf{R}$$

with expected return:

$$\mu_p = \mathbf{w}'\boldsymbol{\mu}$$

and variance:

$$\sigma_p^2 = \mathbf{w}'\boldsymbol{\Sigma}\mathbf{w}.$$

Suppose that $N = 2$: this leads to

$$\mu_p = w_1\mu_1 + w_2\mu_2$$

and

[4] Being $\mathbf{v}$ a vector, $\mathbf{v}'$ denotes its transpose.

$$\sigma_p^2 = \begin{bmatrix} w_1 & w_{1N} \end{bmatrix} \begin{bmatrix} \sigma_{11} & \sigma_{12} \\ \sigma_{21} & \sigma_{22} \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}$$

$$= \begin{bmatrix} w_1\sigma_{11} + w_2\sigma_{21} w_1\sigma_{12} + w_2\sigma_{22} \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}$$

$$= w_1^2\sigma_{11} + + w_2^2\sigma_{22} + 2w_1w_2\sigma_{12}$$

This, however, is still a general formulation. The question is: in what way does an investor choose which amount of each asset (i.e. the weights) to be bought? To do so a target is needed, and the choice for the objective is not unique.

As a first case, consider a goal portfolio mean return, μ_0. Recalling all that has been said about the mean-variance criterion and the efficient frontier, it follows that investors in such situation seek to minimize the risk they are exposed to. Formally:

$$\min_w \mathbf{w}'\mathbf{\Sigma}\mathbf{w}$$

under the two already-mentioned constraints. First, the target mean return:

$$\mu_0 = \mathbf{w}'\boldsymbol{\mu}$$

Second, the weights summing up to 1:

$$\mathbf{w}'\mathbf{1} = 1^5$$

It is then possible to solve this minimization problem applying the method of Lagrange multipliers, and the result is:

$$\mathbf{w} = \mathbf{g} + \mathbf{h}\mu_0$$

with

$$\mathbf{g} = \frac{1}{ac - b^2}\mathbf{\Sigma}^{-1}[c\mathbf{1} - b\boldsymbol{\mu}]$$

$$\mathbf{h} = \frac{1}{ac - b^2}\mathbf{\Sigma}^{-1}[a\boldsymbol{\mu} - b\mathbf{1}]$$

and

[5] $\mathbf{1}' = [1, 1, ..., 1]$

$$a = \mathbf{1}'\boldsymbol{\Sigma}^{-1}\mathbf{1}$$

$$b = \mathbf{1}'\boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}$$

$$c = \boldsymbol{\mu}'\boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}$$

This is a significant result, since that is exactly the mathematical way in which the efficient frontier is built, solving such minimization problem for different levels of expected return.

Notice that, although this simple case can be solved analytically, more complex situations will require numerical optimization techniques.

A second way of obtaining the same final result (i.e. the same efficient frontier) is to fix a target value for the portfolio risk, σ_0. This, by consequence, will shift the optimization problem towards the maximization of the expected portfolio mean return:

$$\max_w \mathbf{w}'\boldsymbol{\mu}$$

subject, again, to two constraints:

$$\mathbf{w}'\boldsymbol{\Sigma}\mathbf{w} = \sigma_0^2$$

$$\mathbf{w}'\mathbf{1} = 1$$

As already highlighted, from here the same result is obtained. Still, it can be useful to have more than one approach to the same problem. For instance, a manager may be asked to attain to a certain risk level. This is the case with managements mimicking a benchmark.

The third and last way of finding the optimal weights is to adopt an approach closer to the utility function theory. In this case, indeed, it is required to explicit the utility given by the portfolio by means of a risk aversion coefficient, λ. What is maximized, now, is a combination of both expected return and risk:

$$\max_w(\mathbf{w}'\boldsymbol{\mu} - \lambda\mathbf{w}'\boldsymbol{\Sigma}\mathbf{w})$$

It follows that the constraints reduce to only one:

$$\mathbf{w}'\mathbf{1} = 1$$

The greater the value taken by λ, the higher the risk aversion and, thus, the higher the penalization given to risky assets.

The efficient frontier is obtained by gradually increasing λ starting from zero.

Lastly, there is an intuitive way to compute the global minimum variance portfolio: it consists in minimizing the portfolio variance without imposing any sort of constraint on the expected return. This makes so that the computation can lead to the minimum value possible, no matter what is the level of remuneration, which is exactly the goal here. Formally:

$$\min_{w} \mathbf{w'\Sigma w}$$

under one unique requirement:

$$\mathbf{w'1} = 1$$

which is solved by:

$$\mathbf{w} = \frac{1}{\mathbf{1'\Sigma^{-1}1}}\mathbf{\Sigma}^{-1}\mathbf{1}$$

Taking as a model a numerical example that can be found in Chapter 2 of Fabozzi *et al.* (2006), it is possible to understand how investors choices vary in response to diversification and short selling constraints. This is done by observing how the efficient frontier modifies its shape:

Figure 2.5 shows that, as the portfolio contains more low correlated assets, the efficient frontier widens. In this case, ef_1 represents the less diversified portfolios while ef_3 is the result of maximum diversification.

To check the goodness of a higher variety of assets it is sufficient to pick an arbitrary risk level, σ_p, and observe how the expected return for the corresponding portfolio increases with diversification. Indeed, it results that $E(r_1) < E(r_2) < E(r_3)$.

It is then possible to generalize this evidence by computing the portfolio variance's upper bound, as follows:

$$var(R_p) = \mathbf{w'\Sigma w} = \frac{1}{N^2}\sum_{i=1}^{N} var(R_i) + \frac{1}{N^2}\sum_{i\neq j} cov(R_i, R_j)$$

$$\leq \frac{1}{N^2}N\sigma^2_{max} + \frac{1}{N^2}(N-1)N \cdot A = \frac{\sigma^2_{max}}{N} + \frac{N-1}{N} \cdot A$$

Here, σ^2_{max} denotes the largest covariance observable among all the assets, while A stands for the average pairwise asset covariance, computed as:

$$A = \frac{1}{(N-1)N}\sum_{i\neq j} cov(R_i, R_j)$$

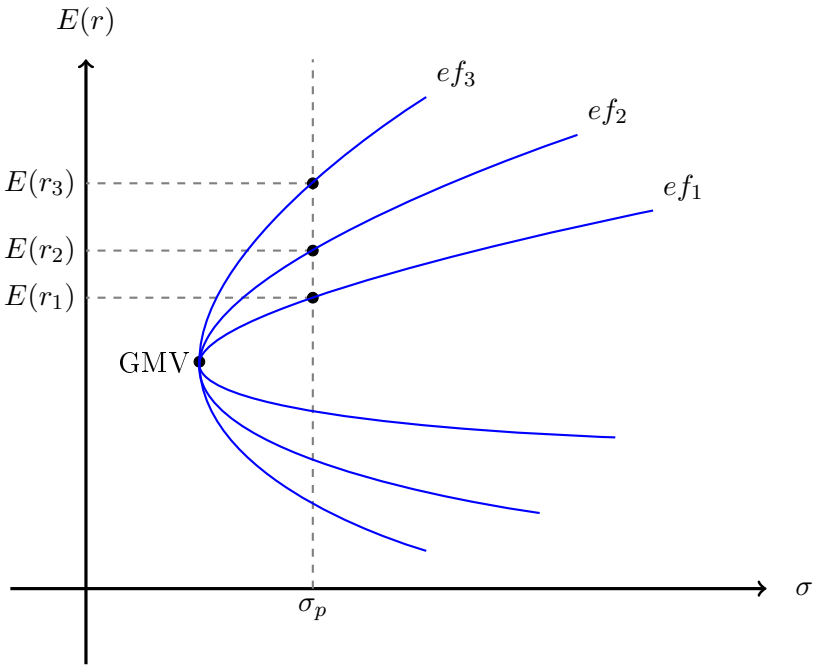


Figure 2.5: Diversification effect on the efficient frontier

Now, if all the variances and covariances exist, this means that they are finite and hence bounded. It follows that A itself is bounded, and that the overall portfolio variance is too:

$$var(R_p) \xrightarrow{N \to \infty} A$$

This result has a strong implication: on one hand, it confirms how diversification is effective in reducing the exposure to risk while holding a portfolio, since such limit says that as we increase the number of assets N the total variance decreases; on the other hand, it highlights a limit in this procedure, since this limit towards which $var(R_p)$ is greater than zero, meaning that it will be not possible to completely eliminate portfolio risk.

Figure 2.6, instead, makes evident how preventing portfolio managers from completing short selling operations has an opposite effect. The constrained curve (ef_c), indeed, is narrower than the unrestricted one (ef_u) and, thus, the unconstrained portfolio return, $E(r_u)$, is greater than the constrained one, $E(r_c)$, when they share the same level of risk, σ_p.

Although it turns out to be advantageous for portfolio construction, short selling is not always an available option and the causes are many: from legal or policy reasons to difficulties in going short on a specific asset due to the market.

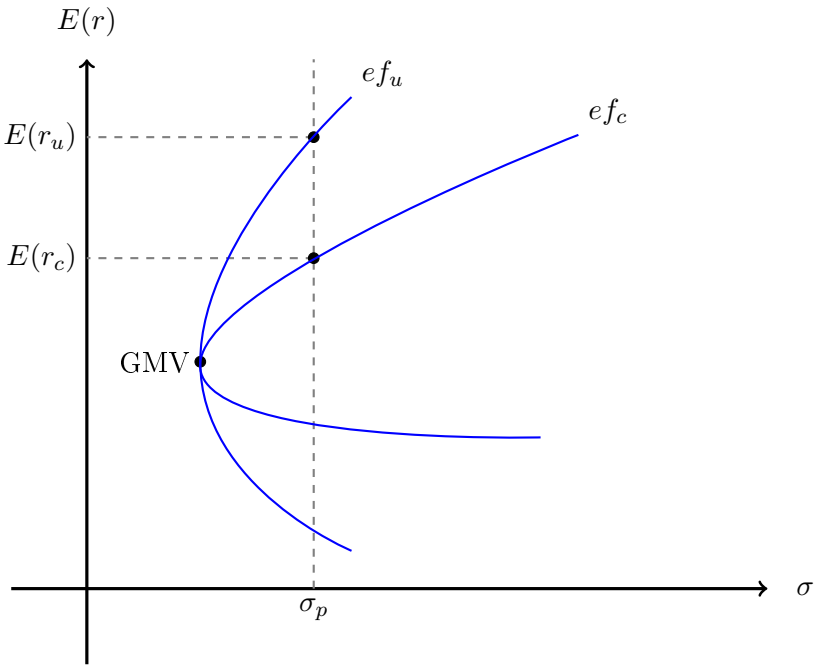


Figure 2.6: Short selling constraint effect on the efficient frontier

Adding a risk-free asset

So far, the analysis focused on portfolios containing only risky assets. However, as shown in many works such as Tobin (1958), Sharpe (1964) and Lintner (1965), the number of efficient portfolios available for investors who can buy a risk-free asset on the market is wider. Hence, it is crucial to examine such situation.

At this end, imagine the market still offers N risky assets but, at the same time, also a risk-free asset with return R_f. This is coupled with the possibility for the investor to borrow and lend at such rate R_f. Notice that this is a strong assumption, since in reality many investors cannot even borrow and lend at the same rate, no matter what its value is. Since the choice is between $N+1$ assets, it follows that now the assumption about $\mathbf{w}' = (w_1, w_2, ..., w_N)$ is no more valid. The risky weights, instead, will sum up to an amount $\mathbf{w}'_R$, while the risk-free asset will be bought for a quantity equivalent to $(1 - \mathbf{w}'_R)$, which can be either positive or negative in this set up. This, by consequence, leads to a modification of the formulas seen for the N risky assets case. In particular, the new portfolio's expected return is:

$$\mu_p = \mathbf{w}'_R \boldsymbol{\mu} + (1 - \mathbf{w}'_R) R_f$$

while the portfolio's variance is:

$$\sigma_p^2 = \mathbf{w}'_R \Sigma \mathbf{w}_R$$

As it can be seen, σ_p^2 is a function of the risky weights but not of the risk-free one. This is possible since, by definition, the risk-free asset is riskless and uncorrelated with other assets.

Having defined those elements, it is again possible to individuate the investor's objective and the subsequent optimization problem to be solved. Say that, for instance, the investor fixes a target mean expected return, μ_0: as before, the goal is to minimize the portfolio's variance. However, pay attention to the variable under consideration. Formally, the minimization has to be operated with respect to the weights of all the assets involved (i.e. both risky and risk-free ones); practically, this corresponds to a minimization with respect to $\mathbf{w}'_R$ only, since the risk-free weight is nothing but a function of the risky weights. The result is:

$$\min_{w_R} \mathbf{w}'_R \boldsymbol{\Sigma} \mathbf{w}_R$$

and the constraint now is:

$$\mu_0 = \mathbf{w}'_R \boldsymbol{\mu} + (1 - \mathbf{w}'_R) R_f$$

Again, the computations are skipped and the final result is presented:

$$w_R = C \boldsymbol{\Sigma}^{-1} (\boldsymbol{\mu} - R_f \mathbf{1})$$

with

$$C = \frac{\mu_0 - R_f}{(\boldsymbol{\mu} - R_f \mathbf{1})' \Sigma^{-1} (\boldsymbol{\mu} - R_f \mathbf{1})}$$

So, as can be seen by the formula, the optimal solution for the minimum variance portfolio composition, w_R, consists in a combination of risky assets ($\boldsymbol{\mu}$) and risk-free asset (R_f). The risky part of the portfolio built in such way takes the name of *tangency portfolio*, and the reason why will be clearer when a graphical representation will be given. For the time being just notice that, as shown in Fama (1970), under ideal assumptions the tangency portfolio should contain all the assets available in the market, each of them in a quantity proportional to their relative market value. That is why it is also called *market portfolio*.

Intuitively, it follows that setting the weights equal to $\mathbf{w}_R^0$, such that $(\mathbf{w}_R^0)'\mathbf{1} = 0$, means creating a risk-free portfolio. On the other hand, $\mathbf{w}_R = \mathbf{w}_R^M$, such that $(\mathbf{w}_R^M)'\mathbf{1} = 1$, translates into the creation of a purely risky portfolio. This intuition, together with the previous computations about the optimal portfolio, can be exploited to actually define a formula for the composition of the market portfolio (which are nothing but $\mathbf{w}_R^M$). From the previous optimization we know that:

$$\mathbf{w}_R^M = C^M \boldsymbol{\Sigma}^{-1}(\boldsymbol{\mu} - R_f \mathbf{1})$$

that, expanding C^M, equals:

$$\mathbf{w}_R^M = \frac{1}{\mathbf{1}'\boldsymbol{\Sigma}(\boldsymbol{\mu} - \boldsymbol{R_f}\mathbf{1})} \boldsymbol{\Sigma}^{-1}(\boldsymbol{\mu} - R_f \mathbf{1})$$

As always, it is possible to create a graphical representation of the possible portfolio combinations among which the investor can choose:

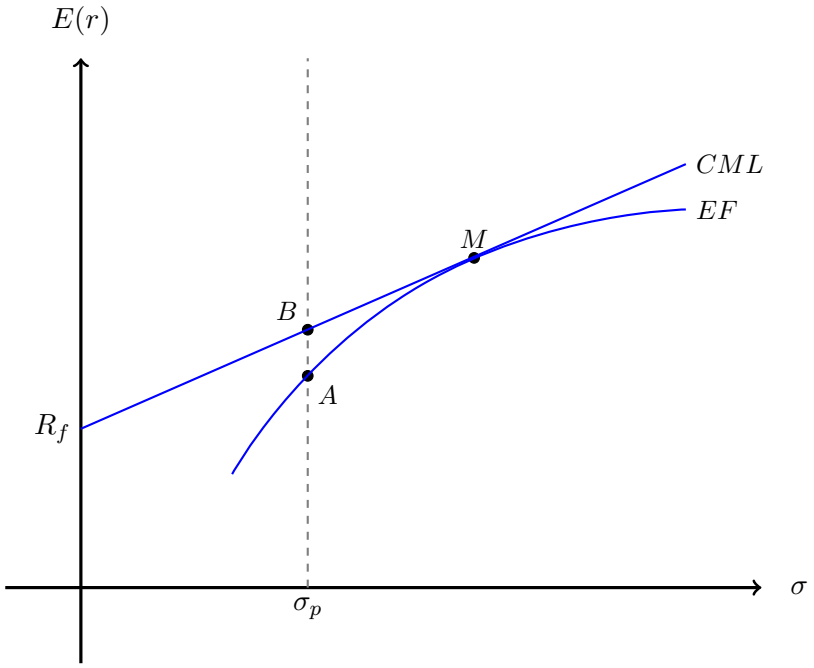


Figure 2.7: Capital Market Line and Efficient Frontier

In Figure 2.7, the efficient frontier (EF) is represented together with a new element: the *Capital Market Line* (CML). While EF depicts the best combinations of risky assets for a risk-averse investor, CML is the representation of all the possible portfolios that can be created by mixing a risk-free assets to the previous N risky ones. In particular, notice how the point of the line touching the vertical axis corresponds to the above mentioned riskless solution ($\mathbf{w}_R^0$), while the point M represents the tangency portfolio ($\mathbf{w}_R^M$). The market portfolio, moreover, represents an important turning point in Figure 2.7: any point of the capital market line lying to its left, indeed, represent portfolios constructed through the purchase of the risk-free asset, while any point to its right is a *leveraged portfolio*, since in that case the investor is borrowing at the risk-free interest rate.

Graphical analysis is a further confirmation of what has been said earlier: purely risky portfolios are inferior to mixed ones. Consider portfolio A, on

the efficient frontier, and portfolio B, on the capital market line: it is well evident that, for a risk-averse investor exposed to a fixed level of risk, σ_p, B would be the best choice available, granting a higher level of expected return.

The fundamental conclusion that all the investors, in order to maximize their portfolio's return at a given level of risk, must hold a combination of the same two elements (a risk-free asset and the tangency portfolio), at percentages proportional to their risk propensity, takes the name of *mutual fund separation theorem*, *two-fund separation theorem* or, simply, *separation theorem.*

Of course, in order to draw the capital market line, it is possible to derive it algebraically. As just said, any investor will hold a certain amount of risk-free assets, w_f, and the remaining part of risky assets, w_M, so that:

$$w_f + w_M = 1$$

and, rearranging:

$$w_f = 1 - w_M$$

Now, since the expected return of the total portfolio is nothing but a weighted average, it follows that it is computed as:

$$E(r_p) = w_f R_f + w_M E(R_M)$$

$$= (1 - w_M) R_f + w_M E(R_M)$$

$$= R_f + w_M [E(R_M) - R_f]$$

At the same time, remembering that R_f and R_M are uncorrelated and that $\sigma_f^2 = 0$, the global portfolio's variance will be:

$$var(R_p) = \sigma_p^2 = w_f^2 \sigma_f^2 + w_M^2 \sigma_M^2 + 2 w_f w_M \sigma_{f,M}$$

$$= w_M^2 \sigma_M^2$$

From here it is possible to compute the standard deviation of the global portfolio:

$$\sqrt{\sigma_p^2} = \sqrt{w_M^2 \sigma_M^2}$$

$$\sigma_p = w_M \sigma_M$$

Rearranging:

$$w_M = \frac{\sigma_p}{\sigma_M}$$

that can be plugged into the above formula for the expected portfolio return, thus to obtain the explicit expression for the capital market line:

$$E(r_p) = R_f + \left[\frac{E(R_M) - R_f}{\sigma_M}\right] \sigma_p$$

In particular, it is important to focus on the term in square brackets, known as the *risk premium*, which corresponds to the slope of the capital market line. At the numerator, the return one would expect from a risky investment in the market (i.e. in the tangency portfolio) is reduced by the certain remuneration coming from the purchase of the risk-free asset. In other words, it represents the excess return an investor could expect when choosing to bear some risk over staying riskless. Such quantity is divided by an element representing the weighted average risk coming from the market. Such division operation makes so that the risk premium is actually a premium per unit of market risk.

Hence, the capital market line formula can be translated into plain words by saying that the remuneration that an investor can expect from a portfolio constructed according to the separation theorem is given by a constant (because riskless) amount, plus a premium for bearing the market risk which equals the amount of risk born times the per-unit cost of it.

So far, two crucial concepts have been exposed: on one hand, the investor's risk propensity and utility function; on the other hand, the efficient frontier and the capital market line, on which the optimal portfolios lie. The natural prosecution to this understanding how those two theories coexist, in order to make the investor choose the best portfolio according to her preferences and characteristics.

As already pointed out, the assumption is that the investor is risk-averse. Starting from here, it is intuitive to say that the optimal portfolio is the one maximizing the expected return for a given level of risk. At the same time, the investor has preferences which are represented by her utility function: in this case, the optimal portfolio is that which, among the feasible ones, ensures the maximum utility level. This concept, first of all, can be very well depicted by putting together Figure 2.3 and Figure 2.7:

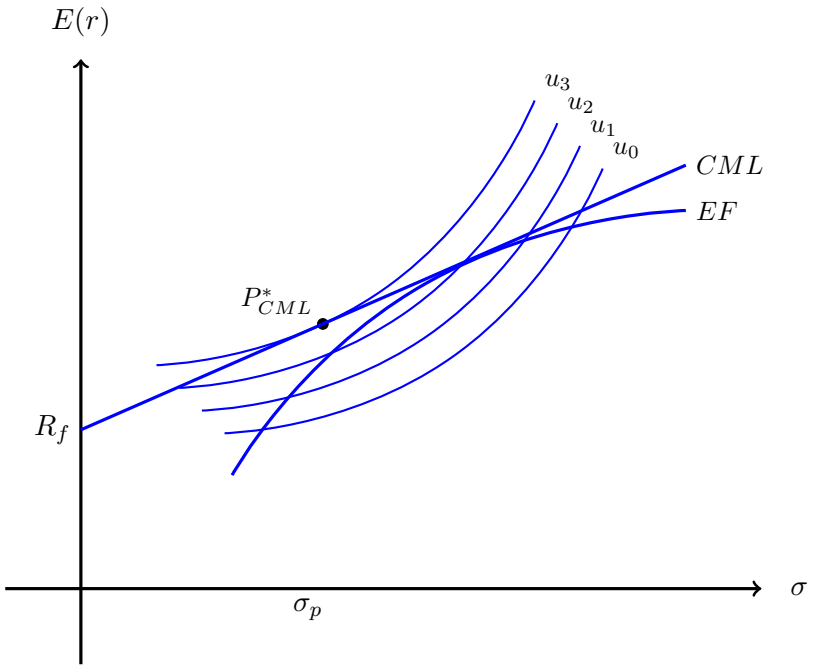


Figure 2.8: Optimal portfolio selection

The investor's choice (P^*_{CML}), then, will be the point of tangency between the capital market line and the highest possible indifference curve (in this case, u_3). In case of absence of a risk-free asset, such utility level couldn't be reached, confirming the goodness of the two-fund separation theorem.

The concepts illustrated up to here represent the very basis of portfolio construction theory. The topic is vast and complex but, to the purpose of this work, this is sufficient. My analysis, indeed, deals with the construction of a purely risky portfolio, as the one mentioned above.

Chapter 3

Pyevolve: a tutorial by examples

Pyevolve is a powerful library devoted to the implementation of genetic algorithms and, in the latest version, genetic programming into the Python programming language. It has been created and developed by the Brazilian software engineer Christian S. Perone, who released the latest version of his work (v0.6rc1, the one I used) on April the 25th, 2010.

Perone's blog about Pyevolve, Python and genetic algorithm programming in general can be found at the link `http://pyevolve.sourceforge.net/wordpress/`, while the Pyevolve documentation can be downloaded or consulted online at `http://pyevolve.sourceforge.net/0_6rc1/`.

To install the library, regardless of the operating system being used, visit `https://github.com/perone/Pyevolve` and download the .zip file. Once this is done, extract its content and reach the extracted folder using the terminal. Here, just enter the command `python setup.py install` and Pyevolve will be automatically installed.

Despite Perone's blog, on the Internet we observe a lack of documentation, discussion and examples about Pyevolve and so, instead of leaving to the reader the task of finding the required information, I will illustrate some relevant features of the library using a series of examples. Those are nothing but the simple programs I wrote while learning to code using Pyevolve myself.

Please notice that the following cannot be, and has not the purpose to be, an exhaustive guide to the whole possibilities offered by Pyevolve.

3.1 Example 1: a matrix containing only zeros

To start off, I will analyze a tiny program very similar to an example given by Perone himself: the purpose is to generate a 20x20 matrix whose elements are all zeros.

Remember that the goal of genetic algorithms is always that of finding the individual (called *genome*) that fits the problem best. In order to formalize my objective I need to define a function, that will be used by the program to check the fitness of each genome created by the algorithm. Here, it is sufficient to proceed as usual when I wish to create a function in Python:

```
def eval_func(genome):
    score = 0.0
    for i in xrange (genome.getHeight()):
        for j in xrange (genome.getWidth()):
            if genome[i][j] == 1:
                score += 0.0025
    return score
```

Through the command `def` I create an evaluation function (`eval_func`) which, as can be seen, is a function of an element here named `genome`. In this case, the genome takes the form of a square matrix having 20 rows and 20 columns. Also notice that I want to use a binary genome, meaning that the only values it can contain are 0 and 1.

Next thing to do is to create the variable `score`, which is the control variable for the whole process: it is this value that influences the work of the algorithm. The goal here is to operate directly on the `score` variable, increasing its value any time a precise condition is satisfied: as soon as one single element of the matrix takes value 1, the function rewards it by increasing the `score` by 0.0025 (this quantity is not accidental, since the matrix contains 400 elements). Notice that, although it may seem so, `genome` is not a Python list. Nonetheless, it encapsulates one, so it is possible to access its element as usual, just denoting their height and width values in square brackets (`genome[i][j]`).

After having analyzed every element of the matrix by means of a double `for` loop (one for the height of the matrix, `i`, and one for its width, `j`), the function will return the total `score` value as the sum of all the ones contained in it.

So far, even if I defined a crucial element in the whole genetic algorithm process, very little specific Pyevolve coding was involved. The next piece of the program will fix this:

```
genome = G2DBinaryString.G2DBinaryString(20, 20)
```

```
genome.evaluator.set(eval_func)
genome.crossover.set(Crossovers.G2DBinaryStringXSingleHPoint)
genome.mutator.set(Mutators.G2DBinaryStringMutatorSwap)
```

Those four strings are the ones that define the characteristics of the individuals that will take part in the evolutionary process.

The very first issue is to define the desired genome shape: as already mentioned, I need a matrix having 20 rows as well as 20 columns, whose elements can take only binary values. In order to do so, it is sufficient to recall the appropriate class from the ones available, thus to create a new instance for it. In this case, the `G2DBinaryString` (which stands for two-dimensional binary string) class will work perfectly. It is then possible to specify the height and width of the newborn instance by writing the desired values in brackets. The name `genome` is assigned to the sample genome.

In the second string, I declare the evaluation function that the algorithm will use in order to do the fitness computations: as already seen, this is the previously created `eval_func`.

At this point, it must be noticed that specifying those two features (shape and evaluation function) is enough information about the genomes for Pyevolve to correctly work. This is true because every class is equipped with a set of default parameters, meant to keep everything as simple as possible. This, of course, does not mean that such parameters are not changeable.

For example, I want to specify a crossover and a mutator, so in the following two strings I recall a Single Horizontal Point crossover (`G2DBinaryStringXSingleHPoint`) and a Swap Mutator (`G2DBinaryStringMutatorSwap`) from the appropriate modules (`Crossovers` and `Mutators`).

So far, the evaluation function and the genomes features have been defined. It's time to move to the genetic algorithm engine, that will actually execute the evolution:

```
ga = GSimpleGA.GSimpleGA(genome)
ga.setMinimax(Consts.minimaxType["minimize"])
ga.setGenerations(200)
ga.evolve(freq_stats = 10)
```

First, I recall the `GSimpleGA` engine from the `GSimpleGA` module, assigning it to the variable `ga`. As for the genome case, this code line would be enough to run the evolution, since any required parameter has been assigned with a default value. Anyway, it can be very useful to modify some of those parameters in order to have a better control over the evolution, or simply to enhance the engine's performances. That's why lines three and four of the last code block are present: the number of generations that the program will

create and analyze is set to `200` and, through the command `freq_stats`, I ask Pyevolve to print the evolution statistics each `10` generations. To clarify what it means, here is one statistic taken from a run of this program:

```
Gen. 0 (0.00%):
   Max/Min/Avg Fitness(Raw) [0.60(0.56)/0.40(0.44)/0.50(0.50)]
```

This represents `Generation 0`, which is the very first population, created based on the genome features discussed above, but not yet subject to any evolutionary process (which will start from `Generation 1`). The `0.00%` value simply denotes the generation progress thus, in this case, it is inevitably zero (and will be `50.00%` for `Gen. 100`, `100.00%` for `Gen. 200`). The six numbers in square brackets give intuitive information about the evolution state: they represent the maximum, minimum and average fitness and, in round brackets, raw score [1]. In this case, the best individual of `Generation 0` has a fitness score of `0.60`, the worst individual scores `0.40` while the average value is `0.50`.

In this regard, notice that the maximum raw score achievable by a genome is 1, when all its elements equal one. However, the purpose here is to obtain a matrix containing 400 zeros and, so, a 0 raw score. Again, it is crucial to be aware of the default Pyevolve settings, which in this case make any program run a fitness/raw score maximization. Since I need a minimization, I have to set the so called `minimaxType` to `minimize`, from the `Consts` module.

Notice that the last code line is not used to just set the statistics frequency, but it actually contains the commad that starts the whole procedure: `ga.evolve`.

Lastly, I want Pyevolve to print the best individual resulting from the evolution (hopefully, a 400 zeros square matrix) and this is achieved through the simple command:

```
print ga.bestIndividual()
```

For space reasons, the final matrix is not represented here.

Go to section A.1 of appendix A to see the complete code.

3.2 Example 2: summing up to 150

The second example is devoted to finding a list of numbers whose total sum equals a defined amount. I chose 20 numbers summing up to 150, but it is

[1] Notice that, in Pyevolve, two different kinds of score coexist: *raw* and *fitness score*. Raw score is nothing but the output generated by the evaluation function, untouched. Fitness score, instead, is the result of raw score's scaling according to the specific selected method.

extremely easy to change such parameters.

The starting point must be, again, the definition of the evaluation function:

```
def eval_func(genome):
    tot = 0.0
    score = 0.0
    for value in genome:
        if value <= 10.0:
            tot += value
        if tot == 150.0:
            score += 1.0
    return score
```

The name of the function is still `eval_func` and its object is still `genome` but, in this case, I need to add the variable `tot` to the computation, in order to achieve my goal. Indeed, the program will examine each element of the first individual and will add it to the `tot` thanks to the command `tot += value`. Only after this first process is completed for all the 20 genome elements the function will move on and analyze the total sum: if it equals 150 the `score` value will jump directly to 1, otherwise it will stay at 0 all the time. The whole mechanism is iterated for each individual of the population.

It is time now to define the genomes features:

```
genome = G1DList.G1DList(20)
genome.evaluator.set(eval_func)
genome.setParams(rangemin = 0.0, rangemax = 10.0,
                 bestrawscore = 1.0)
```

As already mentioned, I want to work with lists of 20 numbers. Those must be unidimensional and, differently from Example 1, not binary. The `G1DList` (one-dimensional string) class perfectly fits the description. Again, a default parameter helps us here, letting the possible genome elements take only integer values, which is good for this example.

In order to limit the computations to a precise range of numbers, I exploit here a feature of Pyevolve which allows the user to set the maximum and minimum possible values contained in each individual. The general command is `setParams`, with which it is possible to manipulate not only `rangemin` and `rangemax` but, as can be seen above, other variables such as `bestrawscore`. I am going to explain its meaning after the next piece of code is presented:

```
ga = GSimpleGA.GSimpleGA(genome)
```

```
ga.setGenerations(1000)
ga.setMutationRate(0.05)
ga.terminationCriteria.set(GSimpleGA.RawScoreCriteria)
ga.evolve(freq_stats = 20)
```

Some functionalities has already been mentioned, such as the selection of the genetic algorithm engine, of the number of generations and of the statistics printing frequency.

What is new here is `ga.terminationCriteria.set`, which is an interesting and useful feature that simply stops the evolution as soon as a desired condition is satisfied. In this example, I wish to have a total amount of 150, corresponding to a `score` value of 1. Thus, in order to stop the engine when 150 is reached, I set the termination criterion to a raw score equal to 1 , through `RawScoreCriteria`. It is not always the best choice to adopt such criteria, but when the ultimate goal is represented by a well defined value, it can help saving time. Think about this program: maybe, 20 numbers that sum up to 150 were found after 200 generations but, without setting a `ga.terminationCriteria`, the evolution would continue for another (useless) 800 generations. If the same happens when the mass of data and computations is much larger, a quite big amount of time can be wasted.

Also, a second new element is present here: `ga.setMutationRate`. It controls the entity of the mutation that each individual is subject to when evolving from a generation to another one, and the higher the value in brackets the greater the mutation occurred.

I end the program by printing the best individual as well as a message displaying the sum of the 20 integer numbers contained in it, so to check the goodness of the evolution process:

```
print ga.bestIndividual()
print ("The sum of the list's elements is"),
        sum(ga.bestIndividual())
```

The outcome appears as follows:

```
- G1DList
        List size:         20
        List:             [10, 10, 8, 8, 10, 0, 8, 4, 1, 8,
                             3, 8, 6, 10, 8, 10, 10, 8, 10, 10]


The sum of the list's elements is 150
```

Go to section A.2 of appendix A to see the complete code.

3.3 Example 3: equations solving

In this section I will jointly illustrate two examples, since they are very similar. They are both equation solvers, with the only difference that the first one operates on a one variable equation while the second one deals with two variables.

The functions are

$$-x^x + 2x^2 + x - 5 = -11$$

and

$$x^4 + xy + y^4 = 1$$

respectively.

As always, I start by defining the evaluation function which, in this case, corresponds to the desired equation.

The one variable case:

```
def eval_func(genome):
    score = 0.0
    x = genome[0]
    if (-(x**x)+(2*x**2)+x-5) == -11:
        score +=1
    return score
```

The two variables case:

```
def eval_func(genome):
    score = 0.0
    x = genome[0]
    y = genome[1]
    if x**4 + x*y + y*4 == 1:
        score +=1
    return score
```

The mechanism is the same used in Example 2: I assign the value 1 to the `score` variable if and only if the equation of interest is solved by the genome's elements, otherwise it stays at 0. To keep everything as neat as possible, before writing the equation inside the function definition, I exploit the fact that Pyevolve's genomes encapsulate a list, thus to assign their elements to the usual variables `x` and `y`.

What follows is the genomes definition.

One variable case:

```
genome = G1DList.G1DList(1)
genome.evaluator.set(eval_func)
genome.setParams(bestrawscore = 1.0)
genome.crossover.clear()
genome.mutator.set(Mutators.G1DListMutatorRealGaussian)
```

Two variables case:

```
genome = G1DList.G1DList(2)
genome.evaluator.set(eval_func)
genome.setParams(bestrawscore = 1.0)
genome.mutator.set(Mutators.G1DListMutatorRealGaussian)
```

Of course, the most evident difference between the two cases is the number of unknowns involved, which influences the genome length. Both are one-dimensional lists (`G1DList`) but their length is `1` and `2` respectively. Anyway, this is not the only disparity in terms of coding: when individuals containing one unique value are used, it is crucial to notice that any crossover operation is not possible. In this case, the useful choice of including a lot of default parameter values in Pyevolve turns into an issue: without further specification, the program automatically applies the genetic crossover principle, trying to split indivisible individuals (one element lists) and causing a crash. To avoid all these problems, it is sufficient to exclude the crossover through the `crossover.clear()` command.

Again, in both cases I apply the `bestrawscore` criterion to stop the engine whenever a solution is found (meaning `score` equating to 1) and I specify a particular mutator, namely a real Gaussian one.

The genetic algorithm code for case one:

```
ga = GSimpleGA.GSimpleGA(genome)
ga.setGenerations(500)
ga.setPopulationSize(1000)
ga.selector.set(Selectors.GRouletteWheel)
ga.terminationCriteria.set(GSimpleGA.RawScoreCriteria)
ga.evolve(freq_stats = 20)
```

and for case two:

```
ga = GSimpleGA.GSimpleGA(genome)
ga.setGenerations(500)
ga.setPopulationSize(2000)
ga.selector.set(Selectors.GRouletteWheel)
ga.terminationCriteria.set(GSimpleGA.RawScoreCriteria)
ga.evolve(freq_stats = 20)
```

The two are almost identical. Commands already seen, such as `GSimpleGA`, `setGenerations`, `terminationCriteria.set`, `evolve` and `freq_stats`, are present and do not need further explanations.

Two new functions are also present. First, by means of `setPopulationSize` I control the number of individuals forming each generation. The two cases have different size values simply because finding a two unknowns solution is a longer process: doubling the number of individuals increases the possibilities of finding a solutions withing the 500 generations limit set. Of course, this will also slow down the whole computation process.

The second novelty here is represented by the selector setting. With the `selector.set` command I tell Pyevolve to apply the `GRouletteWheel` method when selecting the individuals for the evolution process.

Lastly, I print the best individual as usual:

```
print ga.bestIndividual()
```

This will equal 3 in the first case, while it will equal (0,1) or (1,0) in the second case.

Notice that, because of applying the best raw score termination criterion, only one of the two solutions for the second equation can be found.

Go to section A.3 of appendix A to see the complete codes.

3.4 Example 4: functions maximization and minimization

This set of examples deals with maximization and minimization of mathematical functions.

Firstly, I want to maximize a simple function with two constrained variables:

$$f(x, y) = 10x + 5y, \text{ with } 3 \leq x \leq 10 \text{ and } 4 \leq y \leq 7$$

The code starts with:

```
def eval_func(genome):
    score = 0.0
    x = genome[0]
    y = genome[1]
    if x>=3 and x<=10 and y>=4 and y<=7:
        score = 10*x + 5*y
    return score
```

Given that the object I want to maximize is `score`, after having imposed all the necessary constraints by means of an `if` statement, I create such variable and I assign to it a value exactly equal to the objective function. In this way, any different `x` and `y` values will directly influence my control variable.

The genetic coding follows as usual:

```
genome = G1DList.G1DList(2)
genome.evaluator.set(eval_func)
"genome.setParams(bestrawscore = 135.0000, rounddecimal = 4)"
genome.mutator.set(Mutators.G1DListMutatorRealGaussian)

ga = GSimpleGA.GSimpleGA(genome)
ga.setGenerations(500)
ga.setPopulationSize(2000)
ga.selector.set(Selectors.GRouletteWheel)
"ga.terminationCriteria.set(GSimpleGA.RawScoreCriteria)"
ga.evolve(freq_stats = 20)

print ga.bestIndividual()
```

I select a unidimensional genome of length 2, with `eval_func` as the evaluation function and a real Gaussian mutator. Each population will count 2000 individuals and the evolution will stop after 500 generations are created, the selection method is roulette wheel and the program will print evolution statistics every 20 generations.

Notice that two lines are commented, meaning they are excluded from the actual code. This is because they involve the `bestrawscore` termination criterion which, thinking carefully about it, turns out to be inapplicable in this case. Indeed, not only I don't know which values for x and y will maximize my function (even though I can imagine it, because of the constraints), but I don't even know which will be the maximum value for the function. This means that I cannot apply a termination criterion, since I cannot set a proper terminal value.

In the commented code I chose a termination value of 135.0000 only because this particular case is so simple that finding the maximum is intuitive and can be done without the help of a machine (remember, those examples are simple on purpose, they are meant to show how Pyevolve works). Keeping this general, however, prevents me to apply the criterion.

Also, remember that the maximization mode is the default one in Python, so I don't need to specify it.

Not surprisingly, the function's maximum is found at maximum x and y values, which is the point (10, 7). This is the program's outcome:

```
- G1DList
        List size:          2
        List:               [10, 7]
```

I move now to two minimization problems.
The first equation is really simple:

$$f(x) = x^2$$

The second one, instead, looks as follows:

$$f(x) = 1 + x^2 + y^2$$

Differently from the first case I juts illustrated, now I know where the minimums are, so my only concern is about x and y values. The first function is a parabola with minimum value equal to 0, while the second function is a paraboloid which never takes values lower than 1. Both, of course, have no upper limit.
The two evaluation functions are defined as follows:

```
def eval_func(genome):
    score = 0.0
    x = genome[0]
    score = x*x
    return score
```

and:

```
def eval_func(genome):
    score = 0.0
    x = genome[0]
    y = genome[1]
    score = 1 + (x*x) + (y*y)
    return score
```

The construction is the same as seen above: I create the `score` variable with an arbitrary initial value, I give to the genome's elements the corresponding variables names and, lastly, I associate `score` with my objective function.

Here is the code relative to the genomes definition:

```
genome = G1DList.G1DList(1)
genome.evaluator.set(eval_func)
"genome.setParams(rangemin = -100, rangemax = 100)"
```

```
genome.initializator.set(Initializators.G1DListInitializatorReal)
genome.setParams(bestrawscore = 0.0000, rounddecimal = 4)
genome.mutator.set(Mutators.G1DListMutatorRealGaussian)
genome.crossover.clear()
```

and:

```
genome = G1DList.G1DList(2)
genome.evaluator.set(eval_func)
"genome.setParams(rangemin = -100, rangemax = 100)"
genome.initializator.set(Initializators.G1DListInitializatorReal)
genome.setParams(bestrawscore = 1.0000, rounddecimal = 4)
genome.mutator.set(Mutators.G1DListMutatorRealGaussian)
```

In the first case I have only one variable so, as discussed in Example 3, I have to avoid any crossover operation (which is impossible and would result in a crash) through the `genome.crossover.clear()` command. The second equation involves two variables, so this problem does not show up.

My evaluator is still `eval_func` and I select a real Gaussian mutator.

Notice that now I know in advance what is the minimum value each function can take, so it is perfectly nice to apply a termination criterion. The ending raw scores are 0 and 1 respectively, rounded at the fourth decimal in order to have a higher precision in the solution (if they were rounded at the first decimal, for example, the evolution could stop at a raw score of 1.09).

So far, I always worked with simple examples that required nothing more than working with integer numbers. Now, I want to explore the continuous case, so I need to explicit it setting a precise initializator. From the `Initializators` module i pick the `G1DListInitializatorReal` one, which will make the engine work with real numbers.

Lastly, notice that it could be possible to set a minimum and maximum range for the genomes values, in order to reduce the evolutionary possibilities and speed up the whole process. I excluded this feature from the program since the load for the machine is not so big to require such intervention, thus I preferred to keep the computations as general as possible.

I can move to the genetic algorithm definition now, which will be identical for the two cases, so I report it just once:

```
ga = GSimpleGA.GSimpleGA(genome)
ga.setGenerations(500)
ga.setMinimax(Consts.minimaxType["minimize"])
ga.selector.set(Selectors.GRouletteWheel)
ga.setMutationRate(0.05)
ga.setPopulationSize(500)
```

```
ga.terminationCriteria.set(GSimpleGA.RawScoreCriteria)
ga.evolve(freq_stats=20)

print ga.bestIndividual()
```

I am dealing with minimization problems, so is necessary to specify it with the `setMinimax` command, in order to overcome the default setting.

Generations number and population size are both set to 500, I chose a roulette wheel selector and a mutation rate of 0.05, as well as a statistics frequency equal to 20 generations.

The minimum points for the equations are in 0 and in (0, 0) respectively, and those are Pyevolve's outcomes for `print ga.bestIndividual()`:

```
- G1DList
        List size:          1
        List:               [0]
```

and

```
- G1DList
        List size:          2
        List:               [0, 0]
```

Go to section A.4 of appendix A to see the complete codes.

3.5 Example 5: a Black-Scholes application

With this example, I try to apply genetic algorithms to a more realistic and concrete case: given four values out of five needed, I want to maximize (or minimize) a call option price computed with the well-known Black-Scholes formula, which is equal to:

$$C(S,t) = S_t N(d_1) - Ke^{-r(T-t)} N(d_2),$$

where d_1 and d_2 correspond to:

$$d_1 = \frac{ln\frac{S_t}{K} + (r + \frac{1}{2}\sigma^2)(T-t)}{\sigma\sqrt{T-t}}, \quad d_2 = d_1 - \sigma\sqrt{T-t}$$

S denotes the underlying stock price, K is the strike price, r is the risk-free interest rate, emphT stands for the option maturity, σ is the stock volatility and *N(·)* denotes the standard normal cumulative distribution function.

I start the code by setting all the desired variables:

```
K = input("Insert strike price value: ")
sigma = input("Insert sigma value: ")
T = input("Insert time to maturity value: ")
r = input("Insert risk-free interest rate value: ")
mi = input("Insert minimum underlying price value: ")
ma = input("Insert maximum underlying price value: ")
```

As can be seen, I chose to work with a varying underlying stock price, while all the other values are fixed. Through the command `input`, before starting any computation the user can manually insert the needed amounts, as well as selecting the maximum and minimum values for S.

Now that I have all the numbers, I can proceed with the definition of the evaluation function:

```
def eval_func(genome):
    S = genome[0]
    score = 0.0
    if S >= mi and S <= ma:
        d1 = (np.log (S / K) + ((r + (sigma**2 / 2)) * T)) /
                                (sigma * (np.sqrt(T)))
        d2 = d1 - (sigma * np.sqrt(T))
        score = S * norm.cdf(d1) - K * np.exp(-r * (T)) *
                                    norm.cdf(d2)
    return score
```

I identify `genome[0]` as `S`, in order to have a neat formulation, and I limit its possible values to the ones I set above (`mi` and `ma`). After this, I just have to compute `d1` and `d2`, thus to have all the elements I need to calculate the option price (which, in this case, must correspond to my variable `score`).

Here, `np` and `norm` identify the numpy and scipy packages, which have been imported at the beginning of the code, as well as Pyevolve:

```
from pyevolve import *
import numpy as np
from scipy.stats import norm
```

I can now set the genomes features as usual:

```
genome = G1DList.G1DList(1)
genome.crossover.clear()
genome.setParams(rangemin = mi, rangemax = ma)
genome.evaluator.set(eval_func)
genome.mutator.set(Mutators.G1DListMutatorRealGaussian)
genome.initializator.set(Initializators.G1DListInitializatorReal)
```

I want unidimensional individuals with length 1 (`G1DList(1)`), which imply the need of blocking any crossover operation from the engine (`crossover.clear()`). To enhance the performances of the program, I set the range of the possible genomes values thanks to the variables `mi` and `ma` and, as usual, the evaluation function is set. Lastly, I want the engine to operate on real numbers (`G1DListInitializatorReal`) and to utilize a real Gaussian mutator (`G1DListMutatorRealGaussian`).

Moving to the genetic algorithm code:

```
ga = GSimpleGA.GSimpleGA(genome)
ga.setGenerations(150)
ga.setPopulationSize(500)
"""ga.setMinimax(Consts.minimaxType["minimize"])"""
"ga.selector.set(Selectors.GRouletteWheel)"
ga.evolve(freq_stats = 20)

print ga.bestIndividual()
```

Here I work on 150 generations containing 500 individuals each. In this case I am maximizing the call price, so the minimization command (`minimaxType["minimize"]`) is excluded. Of course, to change from one operation to the other one, it is sufficient to delete the inverted commas.

Go to section A.5 of appendix A to see the complete code.

3.6 Example 6: creating a simple portfolio

This example, as well as the following, will approach more closely the true purpose of my work. I will, indeed, illustrate two simple financial markets that implement Pyevolve.

In this first case, I create an environment in which two groups (one buyer and a variable number of sellers) interact following a basic bid and ask mechanism: each of them is associated with a target price, at which the agent is willing to buy or sell its financial product (say, a bond); after that, the buyer starts checking sellers prices, searching for bonds that are cheap enough to match her own buying price. If she can find one, she will buy until her budget is low enough to impede further transactions. The genetic algorithm goal is to create a portfolio containing four bonds whose average price is as near as possible to 100.

To start off, I set the number of buyers and sellers and I create three empty lists that will reveal their usefulness soon:

```
nBuyers = 1
buyersListP = []
buyersListB = []

nSellers = int(input("How many sellers? ")
sellersListP = []
```

Those lists will contain the prices (`buyersListP` and `sellersListP`) and the buyer's budget (`buyersListB`)

The next step is to randomly create the prices and budgets, thus to put them in my lists:

```
for i in range(nBuyers):
    buyerP = round(uniform(105, 110), 3)
    buyersListP.append(buyerP)
    buyerB = round(uniform(400, 410))
    buyersListB.append(buyerB)

for i in range (nSellers):
    sellerP = round(uniform(90, 110), 3)
    sellersListP.append(sellerP)
```

Prices take values between 105 and 110 (for the buyer) and between 90 and 110 (for the sellers) and are rounded at the third decimal. Buyer's budget is contained between 400 and 410. The lists are filled by mean of the `append` command.

Before creating the loop that will manage the transactions (and that will contain the Pyevolve code), it is necessary to create the list that will include the execution prices, as well as the two variables which will contain the buyers price (`buyP`) and budget (`buyB`):

```
exePrices = []
buyP = itemgetter(0)(buyersListP)
buyB = itemgetter(0)(buyersListB)
```

I have now all the elements to define the evaluation function:

```
def eval_func(genome):
    score = 0.0
    x = genome[0]
    y = genome[1]
    z = genome[2]
    w = genome[3]
    if 99.9 < ((x + y + z + w) / 4) < 100.1
    and (x + y + z + w) <= buyB:
```

```
        score +=1
    return score
```

The first `if` condition sets the average price between 99.9 and 100.1. It is necessary to work in this way, instead of writing `((x + y + z + w) / 4) == 100`, because even such little range makes the computation much more effective. Trying to achieve a perfect 100 average, most of the times will result in a failure.

The second `if` condition simply states that the sum of the prices of the four bonds cannot overcome the initial buyer's budget.

A fundamental novelty will be introduced in the following piece of code:

```
setOfAlleles = GAllele.GAlleles()

for i in range(4):
  a = GAllele.GAlleleList(sellersListP)
  setOfAlleles.add(a)

genome = G1DList.G1DList(4)
genome.setParams(allele=setOfAlleles)
genome.evaluator.set(eval_func)
genome.mutator.set(Mutators.G1DListMutatorAllele)
genome.initializator.set(Initializators.G1DListInitializatorAllele)

ga = GSimpleGA.GSimpleGA(genome)
ga.setGenerations(500)
ga.setPopulationSize(200)
ga.evolve(freq_stats=50)

"print ga.bestIndividual()"
```

As evident, a new element appears in this Pyevolve code: *alleles*. In Pyevolve's language, alleles are the instruments which allow the user to make the genomes take only the values contained in a desired set (so far, the only action I illustrated was defining a range through `rangemin` and `rangemax`, which is not as powerful as alleles).

In order to do so, I first create `setOfAlleles`, that will contain all the values I need. Now, exploiting a `for` loop, I assign to all the genome's elements (in this case, 4) the same range of possible of values. Since I want to work only with the sellers prices and no other values, I recall `sellersListP`, thanks to which i create an element here named `a`, that is then added to the set of alleles.

Once this is done, it is important to remember to set the proper parameters (`genome.setParams(allele=setOfAlleles)`), the proper mutator (`G1DListMutatorAllele`) and the proper initializator (`G1DListInitializatorAllele`).

The rest of the code is written as shown in the previous examples.

Once that the algorithm is run, I can exploit the resulting best individual, extracting the optimal prices from it and assigning them to four variables:

```
firstP = ga.bestIndividual()[0]
secondP = ga.bestIndividual()[1]
thirdP = ga.bestIndividual()[2]
fourthP = ga.bestIndividual()[3]
```

Those variables will be useful in the following loops, in order to be sure that the buyer will buy only those bonds with the desired optimal price, and at the same time will not buy bonds whose price is too high:

```
for i in range(nSellers):
    sellP = itemgetter(i)(sellersListP)

    if firstP < buyP and secondP < buyP
    and thirdP < buyP and fourthP < buyP:
        if sellP == firstP or sellP == secondP
        or sellP == thirdP or sellP == fourthP:
            if sellP <= buyP:
                exePrices.append(sellP)
```

Each time one of the optimal prices respects the buyers parameters, it is added to the `exePrices` list and is considered as bought.

Lastly, if `exePrices` contains all the four prices, a series of data will be printed, in order to check the goodness of the operation:

```
print "The initial budget was:", buyB
print "The total expenditure is:", sum(ga.bestIndividual())
print "The execution prices are:", exePrices
print "The average price paid is:", sum(ga.bestIndividual())/4
```

Here is an example of the final outcome:

```
Buyer price: [106.021]

The buyer buys a stock from Seller 1 at price 99.715
The buyer buys a stock from Seller 27 at price 97.942
```

```
The buyer buys a stock from Seller 35 at price 103.059
The buyer buys a stock from Seller 55 at price 99.016

The initial budget was: 406.0
The total expenditure is: 399.732
The execution prices are: [99.715, 97.942, 103.059, 99.016]
The average price paid is: 99.933
```

As can be seen, the budget and average conditions are both satisfied.

Go to section A.6 of appendix A to see the complete code.

3.7 Example 7: creating a simple portfolio, case 2

My last example still exploits the crucial feature of alleles. It is, again, an economic example, even though the assumptions and choices are very simplistic. I imagine a market with 1000 financial products (say, again, bonds) which only have one feature: the paying interest rate. The unique buyer enters the markets to create a portfolio containing 7 bonds and paying an average interest rate as near as possible to the 5%.

At first, I set the number of bonds and create an empty list that will contain them:

```
nIRs = 1000
List = []
```

Next thing to to is to fill it with random values between 0.1% and 20% by means of a `for` loop:

```
for i in range(nIRs):
    ir = uniform(0.001, 0.2)
    List.append(ir)
```

I am ready now to define the evaluation function:

```
def eval_func(genome):
    score = 0.0
    x = genome[0]
    y = genome[1]
    z = genome[2]
    w = genome[3]
    k = genome[4]
    a = genome[5]
```

```
    b = genome[6]
    if 0.04999 < ((x + y + z + w + k + a + b) / 7) < 0.05001:
        score +=1
    return score
```

Again, notice that asking the program to look for an average exactly equal to 0.05 would turn out to be counterproductive while, instead, specifying a range (even one precise at the fifth decimal number, as I am doing here) makes the computations easier without loosing precision.

Remember I want the buyer to trade only the existing bonds, so the default Pyevolve settings would not work here. I need to define a list of alleles, as already seen in the previous example:

```
setOfAlleles = GAllele.GAlleles()

for i in range(7):
    a = GAllele.GAlleleList(List)
    setOfAlleles.add(a)
```

Now, thanks to the appropriate alleles commands, I can define the genomes characteristics and the engine parameters:

```
genome = G1DList.G1DList(7)
genome.setParams(allele=setOfAlleles)
genome.evaluator.set(eval_func)
genome.mutator.set(Mutators.G1DListMutatorAllele)
genome.initializator.set(Initializators.G1DListInitializatorAllele)

ga = GSimpleGA.GSimpleGA(genome)
ga.setGenerations(500)
ga.setPopulationSize(500)
ga.evolve(freq_stats=50)

print ga.bestIndividual()
```

Lastly, I want to check the goodness of the algorithm by computing the portfolio average interest rate:

```
print "The average interest rate is", sum(ga.bestIndividual())/7
```

Here is a sample outcome:

```
- G1DList
        List size:        7
```

```
List:                [0.054825929997157866,
   0.01415768511106838, 0.09281094691153573,
  0.03637390637939188, 0.039290848452887724,
   0.008250611373711315, 0.1042901171654623]


The average interest rate is 0.0500000064845
```

Go to section A.7 of appendix A to see the complete code.

Chapter 4

A first step: real market data and genetic algorithms

This chapter deals with only one of the two main parts that form my work. Here, indeed, the attention is focused on the portfolio strategies only, without any concern about the market and the prices formation. The aim, here, is simply to study the implementation of pyevolve in a market context and check its working when applied to real data, so to have a solid basis for the following step, consisting in its application to an artificial stock market. At this regard, the ystockquote library will serve the purpose of creating an appropriate dataset for the time being.

4.1 The ystockquote library

The ystockquote package can turn out to be a useful tool for anyone working with financial market data. The functionality it offers, indeed, is that of creating an intuitive link between python and the Yahoo Finance database (`http://finance.yahoo.com/`), allowing the user to obtain the fundamental data for any stock present in any of the financial markets that can be found worldwide.

The library has been created by Corey Goldberg, a software developer from Boston, Massachusetts. Follow the link `http://cgoldberg.github.io/` to visit Golberg's web page where any useful link about his work is stored. In particular, the ystockquote page can be found at `https://github.com/cgoldberg/ystockquote`: here everything is needed to run the package can be downloaded.

To install the library (on any operating system), after having installed pip, it is enough to reach the python installation folder through the terminal and, from here, run the command `pip install ystockquote` and pip will automatically take care of the whole installation process.

The way ystockquote works is quite simple and intuitive, so only one small example of its usage will be presented here. First of all, it is possible to obtain single real time data. For example, through the command:

```
import ystockquote
ystockquote.get_price('GOOGL')
```

python will display the real time price for the Google's stock:

```
'508.72'
```

Of course, in order to obtain data for different stocks, it is sufficient to write the proper ticker in place of `GOOGL`.

More interestingly, the same can be done with historical data on an arbitrary period of time:

```
import ystockquote
from pprint import pprint
ystockquote.get_historical_prices('GOOGL','2014-12-01','2014-12-04')
```

will give the following outcome:

```
{'2014-12-01': {'Adj Close': '539.65',
'Close': '539.65',
'High': '548.79',
'Low': '538.62',
'Open': '545.09',
'Volume': '1982600'},
'2014-12-02': {'Adj Close': '538.59',
'Close': '538.59',
'High': '541.85',
'Low': '534.66',
'Open': '539.45',
'Volume': '2073600'},
'2014-12-03': {'Adj Close': '536.97',
'Close': '536.97',
'High': '541.41',
'Low': '535.21',
'Open': '537.50',
'Volume': '1623800'},
'2014-12-04': {'Adj Close': '542.58',
'Close': '542.58',
'High': '542.69',
'Low': '534.89',
'Open': '537.64',
'Volume': '1629900'}}
```

Notice that such set of data, speaking in terms of python elements, is a dictionary and the dates are its keys.

More on ystockquote and on how to manipulate its outcomes will be discussed in the following section.

4.2 A ten-stock portfolio

Since the goal of those programs is to implement the usage of the pyevolve's functionalities in the context of a portfolio selection, the time series of stocks prices under consideration has been chosen to be very short on purpose.

Moreover, to check the correct implementation of the algorithms, an investor who forms her portfolio on a completely random basis runs together with the one exploiting genetic optimization.

To achieve the goal, three python libraries are required:

```
from pyevolve import *
import ystockquote as ys
import numpy as np
```

and, in order to avoid possible problems with floating point numbers operations, the following option must be set with respect to numpy:

```
np.seterr(invalid='ignore')
```

If this is not done, the risk is that an error message of this kind will be displayed:

```
RuntimeWarning: invalid value encountered in true_divide
```

After these preliminary operations are completed, the following step is creating a dataset that the investors can utilize in order to create their expectations about future stocks prices. As already mentioned, this is done by means of ystockquote.

First of all, hence, ten python dictionaries are created, one for each stock taken under consideration:

```
AAPL=ys.get_historical_prices('AAPL','2013-11-04','2013-11-07')
BAC=ys.get_historical_prices('BAC','2013-11-04','2013-11-07')
KMI=ys.get_historical_prices('KMI','2013-11-04','2013-11-07')
PBR=ys.get_historical_prices('PBR','2013-11-04','2013-11-07')
FB=ys.get_historical_prices('FB','2013-11-04','2013-11-07')
GE=ys.get_historical_prices('GE','2013-11-04','2013-11-07')
HAL=ys.get_historical_prices('HAL','2013-11-04','2013-11-07')
```

```
INTC=ys.get_historical_prices('INTC','2013-11-04','2013-11-07')
SDRL=ys.get_historical_prices('SDRL','2013-11-04','2013-11-07')
SD=ys.get_historical_prices('SD','2013-11-04','2013-11-07')
```

Their names simply correspond to the correlated stock's ticker, and for each of them the data refer to the period going from 04/11/2013 to 07/11/2013, which means working with four-days data.

It is worth remembering once again that the `get_historical_prices` command gives the user historical data collected into a python dictionary. By its nature, a dictionary cannot be manipulated as a simple array would be. For example, it is not possible to just recall a particular dictionary's element by specifying its position (e.g. a command of the type `AAPL[0]` will not recall the first element of the dictionary `AAPL`).

However, since such features are extremely useful in the context of the manipulation of stocks data, it is necessary to transform such dictionaries into arrays. Taking as an example the case of the Apple stock, the code that serves this purpose looks like:

```
pricesAAPL = []
a = 0
for key in AAPL:
    a = float(AAPL[key]['Open'])
    pricesAAPL.append(a)
```

First line creates an empty array, destined to contain the stock's prices and, right after that, the variable `a` is created and set equal to zero. Its purpose is to be inserted inside the successive `for` loop.

Notice how the loop is run over a specific element of the dictionary: the key. Since in the case of ystockquote keys correspond to the date of each day, such a construction allows to iterate the loop over the time length previously set (in this case, over all the four days).

The fourth line of this piece of code is the crucial one: it is devoted at sorting the dictionary in order to reach the desired element (i.e. the opening price) of the desired day. This is not enough: such element is transformed into a floating point number through the command `float` and assigned to the variable `a`. At last, the command `append` positions the opening price at the end of the previously created array.

This process, of course, is applied to all the ten stocks of interest.

In order to create investor's expectation, however, this is not enough. Agents, indeed, operate on stocks returns, which need to be manually computed, by means of the formula:

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

To the purpose of using relative instead of absolute price changes, then, the natural logarithm of the return is computed:

$$ln(r_{i,t})$$

For this simple model the assumption is that expected returns, from the point of view of the investors, perfectly coincide with the mean of returns. Moreover, what is of interest is to know the portfolio volatility. To these ends, mean and variance of stocks returns are also computed.

Again, the first step is to create the empty vectors where all the data that is going to be computed will be stored. The Apple case:

```
lnReturnsAAPL = []
meanAAPL = []
varianceAAPL = []
```

and:

```
r = 0
T = len(pricesAAPL)
```

where `r` is the return variable to be used in the following loop and `T` denotes the time length of the period expressed as the number of days, corresponding to the length of any one of the arrays containing assets prices.

Since the focus is on assets returns, ad given the above formula to compute them, the duration of the loop must be thought carefully. It cannot be simply equal to `T`, because this would trigger a feature of python and the way it recalls arrays elements: in general, the command `vector[-1]` refers to a precise element of the array `vector`, which is its last (this makes sense, since the first element stands in position `0`). Referring this to the current situation, the effect would be that for `t=0` the return formula would turn out to be:

$$r_{i,0} = \frac{p_0}{p_3}$$

which makes no sense.

To avoid this kind of error, hence, the loop range is reduced from `(0, T)` to `(1, T)`, and this is how it looks for the Apple case only:

```
for t in range(1,T):
    presentMeans = []

    r = pricesAAPL[t]/pricesAAPL[t-1]
    lnr = np.log(r)
    lnReturnsAAPL.append(lnr)
```

```
muAAPL = np.mean(lnReturnsAAPL)
meanAAPL.append(muAAPL)
presentMeans.append(muAAPL)
varAAPL = np.var(lnReturnsAAPL)
varianceAAPL.append(varAAPL)
```

Its functioning is quite linear. The first thing to be done is the creation of the empty array `presentMeans`, that will contain the mean log-returns of all the ten assets under consideration. Notice that this command is inside the loop, and not before it, for a precise reason: the `presentMeans` vector has only a temporary use, limited to the length of one cycle of the loop. As soon as it starts over again, the utility of the array ceases and can hence be reset, ready to be used again in the new cycle.

After this preliminary step, the return computation can actually begin: `r` is the variable containing the raw stock's return computed according to the formula seen earlier, while `lnr` simply contains its natural logarithm, computed thanks to the numpy library, called by the command `np.` before the operation is explicitly defined. Now that the variable of interest has been computed, it is possible to collect it into the previously created vector: through the `append` function, `lnReturnsAAPL` receives `lnr` in its last position.

The functionalities of numpy are not limited to the computation of logarithms, but allow the user to rapidly perform statistical calculations, avoiding to write down several lines of additional code. This is exactly how the mean (`muAAPL`) and the variance (`varAAPL`) of stock's returns are computed here: `np.mean` and `np.var` functions accept the vector containing the log-returns, `lnReturnsAAPL`, as an argument and perform the required operations. Again, the last passage of the loop consists of positioning the newborn variables inside the appropriate arrays (`meanAAPL` and `varianceAAPL`) by means of the `append` command. All this without forgetting the `presentMeans` vector, that will reveal its utility in a moment.

Before moving on to the portfolio selection piece of code, however, one last computation is left and must be performed. To compute a portfolio's variance, indeed, knowing each stock's variance is not enough and an additional element is required: the covariance matrix. Luckily, numpy has the ability of doing it in place of the user:

```
mat = np.matrix([lnReturnsAAPL, lnReturnsBAC, lnReturnsKMI,
                 lnReturnsPBR, lnReturnsFB, lnReturnsGE,
                 lnReturnsHAL, lnReturnsINTC, lnReturnsSDRL,
                 lnReturnsSD])
cov = np.cov(mat)
```

In order to properly work, the `np.cov` function needs, as an argument,

the matrix containing the elements between which the covariance must be computed. In this case such elements are the log-returns, here organized in a matrix in which each row represents a single assets and each column represents a day. Notice that, while the number of rows is fixed and equal to 10 (since the number of assets does not vary over time), the number of columns increases from 1 to 3 as days go by. The `cov` matrix, on the contrary, has dimension 10x10 in each day, with its off-diagonal elements σ_{ij} denoting the covariance between asset i's and asset j's return and the diagonal elements corresponding to the assets variances.

At this point, all the necessary elements have been created and computed, so it's possible to create the desired portfolio strategies.

First, is defined the behavior of an investor choosing her portfolio composition in a random way. The procedure is extremely simple and consists in just one step: generating 10 pseudo-random numbers that sum up to 1. Each of those values represents the weight of a specific asset of the portfolio. Although its simplicity, this operation may represent a tricky obstacle to overcome, due to the restriction imposed. Formally:

$$\sum_{i=1}^{10} w_i = 1$$

While it is immediate to generate free pseudo-random numbers, making them sum up to a specific value is less. In this particular case, since the goal value is 1, a scaling operation can be performed: ask python to generate 10 pseudo-random numbers, no matter what their range is, then sum them and divide each of them by their total sum. The ratios will sum up to 1. In code, for example:

```
a = np.random.random(10)
b = sum(a)
for i in range(10):
    a[i] = a[i]/b
```

This makes so that the vector `a`, in the end, sums up to one.

A second way of accomplishing the same task, and the one I chose for the example, is that of exploiting the properties of the Dirichlet distribution:

```
randWeights=np.random.dirichlet(np.ones(10),size=1)[0].tolist()
```

Thanks to the random numpy's functionalities, the list `randWeights` is created as follows: `np.random.dirichlet` is the numpy's command to draw samples from the Dirichlet distribution, where its first argument represents the parameter of the distribution (which must be of dimension 10 in the case of a sample of length 10) while its second one simply denotes the number

of samples to draw (in this case 1 sample of size 10). Finally, the command `tolist()` is needed to transform the resulting numpy array into a python list.

The interest of the example, other than the weights themselves, is focused on the return and risk featured by the portfolio, so they are computed as follows:

```
expRandReturn = 0
for i in range(len(randWeights)):
    expRandReturn += randWeights[i] * presentMeans[i]

randRisk = 0
    for i in range(len(randWeights)):
        for j in range(len(randWeights)):
            randRisk += randWeights[i]*randWeights[j]*cov[i][j]
```

In both cases, a `for` loop is used to increase the value of a variable, corresponding respectively to the portfolio's expected return (`randReturn`) and variance (`randRisk`). In the first case, the loop must represent a simple summation over the weights, which have to be multiplied by the corresponding asset's expected return in order to satisfy the equation:

$$E_t[r_{p,t+1}] = \sum_{i=1}^{10} w_{i,t} E_t[r_{i,t+1}]$$

The variance case, instead, is a little more complex, since the summation is a double one. In programming language, this corresponds to a double `for` loop, that allows to represent such equation:

$$\sigma^2_{p,t} = \sum_{i=1}^{10} \sum_{j=1}^{10} w_{i,t} w_{j,t} \sigma^2_{ij,t}$$

where $\sigma^2_{ij,t}$ denotes the (i, j) element of the covariance matrix, `cov`.

The only thing which is left to do, at this point, is the definition of the portfolio strategy based on genetic algorithms. This, actually, can take various forms depending on the specification of the evaluation function. The possibilities are various; here, four cases are presented.

First, a simple return maximization is set. Notice that this is completely risk-unrelated, which makes little sense in real life. The code that defines the objective function is the following:

```
def eval_func(genome):
    score = 0.0
    ret = 0.0
```

```
    for i in range(len(cov[:,1])):
        ret += genome[i] * presentMeans[i]
    if 0.99 < sum(genome) < 1.01:
        score = ret
    return score
```

To start off, return (`ret`) and score (`score`) value are set equal to zero. After that, a loop is set up to compute the portfolio return exactly in the same way has been shown above for the random case. The only, crucial difference is that assets weights are represented by genomes (`genome[i]`), so that the algorithm can work on them with all the genetic operators illustrated in section 1.6. Again, the only constraint is that the weights must sum up to one. However, as already seen in section 3.6, it is not convenient to set such constraint in an extremely precise way (i.e. `==1`), since it would make computations extremely long and difficult to accomplish. Hence, a little accuracy must be sacrificed in favor of computational speed, and this translates into the definition of a range inside which the sum of the weights must fall: `(0.99, 1.01)`. Notice how such constraint appears as part of an `if` condition: this ensures that any solution created by the algorithm which does not satisfy the summation requirement will be immediately filtered and blocked, so that only fitting solutions will be considered. If this is the case, the return can be transferred to the score variable (`score = ret`), in order to be further analyzed by the algorithm.

The second case can be seen as complementary to the first one, since here the only goal of the algorithm is to minimize the portfolio's variance:

```
def eval_func(genome):
    score = 0.0

    for i in range(len(cov[:,1])):
        for j in range(len(cov[1,:])):
            score += genome[i] * genome [j] * cov[i][j]
    return score
```

Again, what I did here is just defining the formula for the computation of risk, as seen above in the random case. The double summation implies a double `for` loop, and here the `score` value is immediately computed as the variance. The reason for it is that, differently from what has been done above, here the algorithm is free of choosing any weight that may fit, without any restriction on their summation up to one. This is possible only because, further in the code, a scaling operation is present so to make the weights ultimately satisfy the constraint.

Third and fourth trials feature an objective function containing both return and risk, put in relation by means of simple mathematical operations.

One is a difference:

```
def eval_func(genome):
    score = 0.0
    ret = 0.0
    risk = 0.0
    for i in range(len(cov[:,1])):
        ret += genome[i] * presentMeans[i]
    for i in range(len(cov[:,1])):
        for i in range(len(cov[1,:])):
            risk += genome[i] * genome [j] * cov[i][j]
    score = ret - (risk)
    return score
```

while the other one is a ratio:

```
def eval_func(genome):
    score = 0.0
    ret = 0.0
    risk = 0.0
    for i in range(len(cov[:,1])):
        ret += genome[i] * presentMeans[i]
    for i in range(len(cov[:,1])):
        for i in range(len(cov[1,:])):
            risk += genome[i] * genome [j] * cov[i][j]
    score = ret/risk
    return score
```

In both cases return (`ret`) and risk (`risk`) are computed exactly as above. The only difference is in the definition of the score value, which no more merely corresponds to one of the two variables but, instead, is a combination of the two, either `ret-risk` or `ret/risk`.

After the evaluation function is set, it is necessary to define all the genetic parameters required by pyevolve:

```
genome = G1DList.G1DList(10)
genome.evaluator.set(eval_func)
genome.mutator.set(Mutators.G1DListMutatorRealGaussian)
genome.initializator.set(Initializators.G1DListInitializatorReal)
```

Genome's shape is that of a vector of length 10, the objective function is one of the four ones listed above, mutator is Gaussian and the initializator will generate real numbers.

Moreover:

```
ga = GSimpleGA.GSimpleGA(genome)
ga.setGenerations(500)
"""ga.setMinimax(Consts.minimaxType["minimize"])"""
pop = ga.getPopulation()
pop.scaleMethod.set(Scaling.SigmaTruncScaling)
ga.setPopulationSize(200)
ga.evolve(freq_stats=50)
```

The number of generations over which the evolution will run is equal to 500, populations will contain 200 individuals each and statistics will be printed every 50 generations. The minimization command line (`ga.setMinimax`) is commented here, since most of the evaluation functions require a maximization. In the second case, however, the goal is to minimize the portfolio's variance, which means that the code must be executed.

Notice fourth and fifth lines, which are devoted to setting up the sigma truncation scaling method. This is necessary because it is the only scaling method allowing to deal with functions with negative results, that would otherwise result in an error that would stop the program.

Last thing to do, in order to be able to compare genetic and random results, is to compute risk and expected return for the genetic investor:

```
best = ga.bestIndividual()
genWeights = []

for i in range(lenbest)):
    a = best[i]/purplesum(best)
    genWeights.append(a)

expGenReturn = 0
for i in range(len(genWeights)):
    expGenReturn += genWeights[i] * presentMeans[i]

genRisk = 0
for i in range(len(genWeights)):
    for i in range(len(genWeights)):
        genRisk += genWeights[i] * genWeights[j] * cov[i][j]
```

The computations are the same as performed with reference to the random case, however an additional preliminary operation is required. As said, indeed, only the first case (profit maximization) works under the explicit constraint requiring the weights to sum up to one. All the other cases, instead, need a scaling, which is performed in the first lines of the previous block of

code. The optimal solution found by the genetic algorithm is assigned to the list `best` and an empty array, `genWeights`, is created. The following loop, identical to the one illustrated earlier when talking about the processes to generate pseudo-random numbers summing up to one, fills the array with scaled values, which correspond to the investor's weights.

After that, `expGenReturn` and `genRisk` can be computed as usual and everything is printed for an immediate comparison:

```
print "Genetic portfolio's weights are: ", genWeights
print "Genetic portfolio's expected return is: ", expGenReturn
print "Genetic portfolio's variance is: ", genRisk

print "Random portfolio's weights are: ", randWeights
print "Random portfolio's expected return is: ", expRandReturn
print "Random portfolio's variance is: ", randRisk
```

Talking about the results, the case of pure return maximization turns out to be a particular one. Without any risk concern, indeed, even before running the program it is trivial how the portfolio composition will be the less diversified possible one, because the highest return in this setup is provided by the highest-return asset alone. Any outcome of the program confirms the theory; for example:

```
Genetic portfolio's weights are:  [0.0, 0.0, 0.0, 0.0, 1.0,
                                   0.0, 0.0, 0.0, 0.0, 0.0]
Genetic portfolio's expected return is:  0.00893328992234
Genetic portfolio's variance is:  0.00343783494143

Random portfolio's weights are:  [0.302, 0.056, 0.057, 0.093,
                                  0.003, 0.209, 0.074, 0.043,
                                  0.090, 0.073]
Random portfolio's expected return is:  0.000965337510899
Random portfolio's variance is:  8.99122593975e-06
```

Random weights are rounded for space reasons. The data refer to 06/11/2013.

It's evident how, in this case, the genetic portfolio consists of the Facebook stock only (the fifth one in the list) since it was the most performing in that day, so that its total expected return perfectly coincides with the stock's one. As a consequence, genetic strategy outperforms the random one. Nonetheless, as predictable, the price for a higher expected return is a higher volatility, as can be seen from the variance data above.

The second case deals with pure variance minimization, and those are some results, still referring to 06/11/2013 (with rounded weights):

```
Genetic portfolio's weights are:  [0.078, 0.225, 0.012, 0.183,
                                   0.0, 0.032, 0.238, 0.038,
                                   0.038, 0.157]
Genetic portfolio's expected return is:  0.00109410249825
Genetic portfolio's variance is:  1.74065270661e-19

Random portfolio's weights are:  [0.245, 0.029, 0.091, 0.024,
                                  0.128, 0.256, 0.003, 0.125,
                                  0.087, 0.016]
Random portfolio's expected return is:  0.000714953591461
Random portfolio's variance is:  0.000177451346657
```

It's interesting to notice how not only the genetic weights this time take values that differ from `1.0` and `0.0`, but how the only stock that is not bought in order to achieve the goal the Facebook one.

It's more than evident how the optimization worked, resulting in a genetic variance about 10^{-5} times smaller than the random one. However, the expected return is still higher in the genetic case, and this condition seems to persist in the majority of the runs of the program.

Third configuration of the portfolio strategy adopts an objective function of the form `ret-risk`, giving the following results for 06/11/2013:

```
Genetic portfolio's weights are:  [0.235, 0.0, 0.0, 0.0, 0.059,
                                   0.0, 0.235, 0.0, 0.235, 0.235]
Genetic portfolio's expected return is:  0.00633659571821
Genetic portfolio's variance is:  3.19084493153e-08

Random portfolio's weights are:  [0.025, 0.074, 0.381, 0.049,
                                  0.035, 0.061, 0.010, 0.039,
                                  0.320, 0.006]
Random portfolio's expected return is:  -0.000236822860243
Random portfolio's variance is:  0.000154533234128
```

Weights are rounded again.

First of all, adopting a slightly more sophisticated genetic strategy turns out to be profitable, since expected returns are higher on the sample dataset, while variances are lower. What may catch the reader attention is the fact that, looking at the genetic weights, it seems like the algorithm composed the portfolio following a sort of pattern: many stocks are not bought at all, while four of them have the same (rounded) weight, equal to `0.235`.

Lastly, the fourth strategy changes the form of the evaluation function into a `ret/risk` one. Referring to 06/11/2013 and rounding the weights, those are the results:

```
Genetic portfolio's weights are:  [0.224, 0.138, 0.028, 0.010,
                                   0.0, 0.021, 0.139, 0.185,
                                   0.0358, 0.222]
Genetic portfolio's expected return is:  0.00312323008051
Genetic portfolio's variance is:  4.82808779935e-20

Random portfolio's weights are:  [0.044, 0.083, 0.293, 0.015,
                                  0.115, 0.092, 0.246, 0.080,
                                  0.007, 0.024]
Random portfolio's expected return is:  -1.51911613113e-05
Random portfolio's variance is:  0.000119874025673
```

Again, genetic algorithms outperform the random strategy and, with respect to the previous case, the resulting portfolio features a lower volatility but, also, a lower expected return.

Please notice that the purpose of those examples is not that of a deep and rigorous search for an optimal portfolio selection method, but is actually that of a technical implementation of various elements. This is the reason why the results presented are partial.

In conclusion, such results seem to confirm the goodness of the implementation of the strategies in the context of a stock market, giving the possibility of moving on with the whole work.

To see the full code, go to Appendix B.

Chapter 5

Artificial stock market

This chapter presents the core of my work. Here, the code defining the artificial stock market is developed, together with the implementation of pyevolve's functionalities in order to create portfolio strategies based on genetic algorithms.

Before moving on to the illustration of the code, it must be stressed out that the artificial market suffers from a series of simplifying assumptions.

First of all, there exist no firm behind the stocks being traded, and this translates into the absence of dividend payments. At the same time, the fact that the financial market is not surrounded by a real economy means that the investors have no access to any kind of information related to the stocks. They cannot know weather a firm is going to start a new business or shut down a factory, weather a firm will pay dividends or will retain them, weather the government is going to increase taxes or is going to incentive new assumptions. This inevitably reflects on the investors' ability to create an expectation about future price movements.

However, the lack of information has a second function: it makes so that all the individuals find themselves in the same informative situation, something that can be compared to the *strong efficient market hypothesis*, case in which every trader can make no better assumption, about the future stocks return, than its expected value. This is exactly the logic followed in the program: individuals form their expectations based on returns first and second moments only, and the stock prices change in a totally random way.

Another aspect to be aware of is that every trader has the same budget in every time period: there exists no small household as well as no institutional investor. The only exception is represented by the presence of a market maker, having the duty of ensuring market liquidity.

Furthermore, traders incur in no transaction costs and the market features no riskless bond, focusing the analysis exclusively on the formation of a risky portfolio.

5.1 Preliminary operations

As always, the very first block of the code is devoted to import all the necessary libraries and set the preliminary conditions. Hence, the `matplotlib`, `numpy`, `pickle`, `pyevolve`, `text`, `random` and `time` libraries are called to work. Moreover, as already seen in section 4.2, it is necessary to add one more line to avoid error messages related to problems with floating point numbers operations, as well as divisions by zero:

```
import matplotlib.pyplot as plt
import numpy as np
from operator import itemgetter
import pickle
from pyevolve import *
from random import *
import time
np.seterr(divide='ignore', invalid='ignore')
```

To clarify the meaning of the last line of code above: the aforementioned problems can actually occur and, with the inclusion of such command, they will keep occurring. What will not happen is the display of the related error messages. Although this may seem an easy shortcut to avoid facing those errors, what is going on is actually under control: such problems arise because of the precise design of some sections of the program, hence they are perfectly known to exist. The point here is that, even if present, those problems do not affect in any way the correct functioning of the program, and this is why the choice is that of simply hiding the related error messages. Later in the presentation of the code, when the lines in question will be encountered, the whole concept will be made clearer.

At this point, all the necessary elements are present and it is possible to start designing the actual program.

First of all, a technical operation that is needed in the process of storing all the relevant data produced by each run of the market:

```
timestr = time.strftime("%Y%m%d_%H%M%S"))
```

The goal of such operation is to keep track of the results obtained in a precisely ordered fashion. The element `timestr`, indeed, is a string containing the date and the time in which the string itself is created. This is achieved by means of the `time` function `strftime`, whose arguments correspond to the desired elements to be displayed as a string. Hence, `Y` stands

for year, `m` represents month and `d` is the day; `H`, `M` and `S` respectively denote hours, minutes and seconds.

To make this clearer, here is a sample outcome:

```
'20150120_143918'
```

Its meaning is easy to interpret: the string has been created on January the 20th, 2015, at 14:39:18.

All this is useful since it helps collecting data sorted by date and time. The way the string `timestr` is implemented into such process will be discussed later.

Another crucial passage to operate is that of determining the market size by setting a series of quantities:

```
nRandom = 5
nEqually = 5
nMomentum = 5
nReversal = 5
nHigh = 5
nLow = 5
nSmall = 5
nBig = 5
nGenetic = 1

nTotal = (nRandom + nEqually + nMomentum + nReversal +
          nHigh + nLow + nSmall + nBig + nGenetic + 1)

nShares = 5

budget = 1000
mmBudget = round(uniform(900000, 1100000), 3)
```

The first variables are devoted to define the number of traders that will operate in the artificial market. As an example: `nRandom` for traders buying and selling following a random strategy, `nGenetic` for those exploiting genetic algorithms in order to determine their portfolio composition. All the other values will be illustrated when each trading strategy will be discussed. Not surprisingly, the value `nTotal` denotes the sum of all such quantities, in order to determine the total amount of traders that will take part to the market operations.

Notice that, in addition to the traders categories just mentioned, the program features one more of them: there also is a market maker. Since it is a single entity, however, it doesn't need a variable such as the ones above.

Moreover, while computing `nTotal`, it is necessary to add `1`, to account for the market maker.

Moving on, `nShares` controls the amount of different stocks to be traded in the market, while `budget` is a variable to be assigned to each trader in order to determine the total amount of stocks that will be possible to buy. It is not a proper budget, but since the goal of the variable is very similar, I called it that way.

The same logic is followed by `mmBudget`, which represents the budget assigned to the market maker, which must be higher in order to ensure liquidity.

The next, fundamental step is to create the vectors that will contain all the main data resulting from market operations, which will also be the object of study and further analysis:

```
exePrices = []
for i in range(nShares):
    exePrices.append([])

exeQuant = []
for i in range(nShares):
    exeQuant.append([])

officialPrices = []
for i in range(nShares):
    officialPrices.append([])

lnReturns = []
for i in range(nShares):
    lnReturns.append([])

means = []
for i in range(nShares):
    means.append([])

variances = []
for i in range(nShares):
    variances.append([])

portfolios = []

oldWeights = []

initialPrices = []
for i in range(nShares):
```

```
    initialPrices.append(round(uniform(45, 145), 3))

initialQuant = []
for i in range(nShares):
    initialQuant.append(int(uniform(180000, 220000)))
```

Please notice that, except that for `portfolios` and `oldWeights`, all the vectors are not simply created as empty, but they undergo a quick procedure that defines their particular structure. Each of them, indeed, contains data referring to all the different stocks exchanged in the market. To each of the shares, and for each variable of interest (execution prices, execution quantities, official prices, logarithmic returns, mean returns and variances of returns, initial prices and quantities of stocks), a separate vector is associated. What the code does, hence, is to define such a structure, creating a situation in which further lines of code simply need to fill such empty structure with data.

To better understand this, consider the first case from the above ones: at first, a simple empty vector `exePrices` is created, but this is enough to meet the needs of the work. Hence, through a `for` loop iterated a number of times equal to `nShares`, a new empty vector is put inside the original `exePrices` vector by means of the `append` function, one for each stock.

What results from such process is an array containing a list of sub-arrays that are ready to store data in a well organized fashion.

For further clarity, this is the outcome once the loop finishes its work (with `nShares = 5`):

```
[[], [], [], [], []]
```

Here, `exePrices[0]` will contain execution prices for stock 1 only, `exePrices[1]` will store those related to stock 2 only, and so on.

All the code shown up to here serves the scope of setting up the world with regard to its main, general features. From this point on, instead, what happens is the characterization of all the actions performed during each of the market cycles.

This is exactly the reason why the program is continues with a very big `for` loop, which encompasses a bit less than the whole code:

```
for t in range(10):

    print "CYCLE ", t+1
    print " "
```

The range of the loop coincides with the time span over which data are created and collected and, of course, every loop cycle represents a market

cycle. To make data displayed on screen easier to sort, the first thing that is printed is the number of the cycle being run, considering that Python convention is to start counting from 0 (hence the choice of `t+1`).

Since traders and stocks have already been defined above, what is left to do in order to fully characterize a working market is creating a demand and a supply for each of the goods being exchanged in it. In particular, each one of the active traders, no matter what category it belongs to, has to be associated to quantities and prices both demanded and offered. However, since each of the trader types features different needs, goals and *modus operandi*, such operations need to be performed separately for each category.

5.2 Traders

As already seen above, traders in the market are of various types, each of which features a unique process aimed at determining the ideal portfolio composition. This is, actually, the only part of the code that differs from a traders category to another. Everything else is common to all traders classes. A separate code, however, is devoted to the market maker characterization, because of its peculiar features.

Moving on to the code, the very first element I care about is the portfolio composition. To obtain the final result, two distinct elements are needed: one is the vector containing the portfolio's shares related to each single market cycle, which represents the goal of the trader for the present time; the other one is the array carrying the information relative to the past portfolio weights, which is the starting point for the present time strategy.

The problem with the latter vector, however, is that at time `0` there exists no old portfolio to look at in order to obtain past information. This is why the traders code begins with such lines:

```
if t == 0:
    for i in range(nTotal):
        if t == 0:
            w = []
        for j in range(nShares):
            w.append(0.0)
        oldWeights.append(w)
```

The very first `for` loop is necessary in order to iterate the process that defines a trader's features over all the desired individuals (number which corresponds to `nTotal`).

What follows, instead, takes care of the issue illustrated above: `if` the current market cycle is the first one (which is equivalent to say `t == 0`),

and hence there exists no past portfolio, traders will consider the weights assigned to each of the stocks present in the market as equal to zero. Such information is contained in the `w` vector, which is created as an empty one and further filled with zeros (one for each share traded, `nShares`) through a second `for` loop together with the usual `append` function.

In the end, `w` itself is attached to the general list `oldWeights`.

An extremely similar set of commands is the applied to individuals' portfolios. At this stage, of course, traders hold empty portfolios, but it is important anyway to create them:

```
for i in range(nTotal - 1):
    ptf = []
    for j in range(nShares):
        ptf.append(0.0)
    portfolios.append(ptf)
```

If and only if the traders finds itself in the very first market cycle, they are asked to fill the empty array `ptf` with a number of `0.0`s equal to the number of shares, `nShares`. Once this is done, `ptf` is appended to the `portfolio` vector, which is deputed to contain all traders' portfolio compositions.

Notice that, differently from the `oldWeights` case, the `for` loop now is shorter: it is indeed iterated a number of time equal to `nTotal - 1`. This is so because, in this case, the market maker's portfolio is not to be generated. It will be dealt with separately.

With `5` traders and `5` shares, at the beginning of cycle `0`, `portfolios` will have such appearance:

```
[[0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0],
 [0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0],
 [0.0, 0.0, 0.0, 0.0, 0.0]]
```

At this point, before further specifying the general traders features, it is time to define the different portfolios compositions, specific to each traders category.

Random traders

The very first class to be characterized is that of random traders, which is important because it somehow takes the same role that control groups have in a scientific control. Their behavior, based on no particular strategy at all, is indeed the very first basis for other agents' performance evaluation: if even a total random strategy, in the long run, is better than the investing approach under exam, the latter can be intended as performing really poorly.

To do so, the first step is that of calling a `for` loop that will iterate the following operations over all the random traders:

```
for i in range(nRandom):
```

It is now possible to create the empty array that will contain the percentage values and, of course, generating those values themselves:

```
we = []
w = np.random.dirichlet(np.ones(nShares), size=1)[0].tolist()
```

As already seen in section 4.2 while illustrating how the `ystockquote` library works, to accomplish the second task numpy's `random` module is exploited by calling the `dirichlet` function, which allows the user to draw samples from the Dirichlet distribution. This serves the scope since the generated numbers will sum up to 1, exactly as the components of a portfolio need to do. As already mentioned, the first argument of the function denotes the parameter of the distribution, which must correspond to the number of observations we want to generate (`nShares`). The second argument indicates the amount of samples to be drawn, which in that case corresponds to `1`. Moreover, in order to obtain a classical Python's list, the `tolist()` function is called to modify the nature of the resulting numpy's array.

The resulting vector `w`, in the case of `nShares = 5`, has this kind of appearance:

```
[0.3341419114301873, 0.15451390227707137, 0.17003701002892427,
 0.2655979078327948, 0.07570926843102233]
```

with

```
sum(w) == 1
```

resulting in:

```
True
```

Given that the trader operates over several market cycles, however, those percentages must be seen as variations with respect to the old composition of the trader's portfolio (`oldWeights`). This is accomplished as follows:

```
for z in range(len(w)):
    w[z] = int(w[z] * budget)
    we.append(w[z] - oldWeights[i][z])
```

The `for` loop operates over all the elements of `w` in order to modify them and make them suitable with the markets needs. Allowing traders to buy and sell quantities exactly equal to the portfolio shares would make the volume

of trading quite small, and that's why each `w`'s element is first multiplied by the first element of the `trader` vector (i.e. `budget`) and, secondly, only the integer part of the resulting number is considered, in order to avoid demand and supply of portions of stocks.

Once this is done, the actual quantities to be traded are computed as the difference between the new `w[z]`s, computed as above, and the old amounts of stocks held by the trader (`oldWeights[z]`). Such differences are appended to the vector `we`, which is reset and made empty at each market cycle.

At last, `we` itself is appended to the general vector `weights`:

```
weights.append(we)
```

Equally weighted traders

The second traders category to be defined is another simple one, if considered from a portfolio strategy point of view. The assets choice process, indeed, is almost absent here: any share traded in the market is kept in the same amount. In formula:

$$\omega_{i,t} = \frac{1}{N}$$

with $\omega_{i,t}$ being portfolio's share dedicated to asset i at time t and N denoting the total number of assets to be included inside the portfolio.

At first, the usual loop is needed:

```
for i in range(nEqually):
```

After that, the very first command to be run is the one creating the empty array that will contain the weights:

```
we = []
```

together with a second empty vector is needed:

```
w = []
```

It will contain the portfolio percentages referring to each stock, that will alter be transformed into quantities representing the amount of each stock to be bought.

Now that both vectors are at disposal, it is time to fill them. The first one to be manipulated is `w`:

```
for n in range(nShares):
    w.append(1./nShares)
```

By means of a `for` loop, running over all the stocks present in the market, the list `w` is increased by a constant quantity, corresponding exactly to the $\omega_{i,t}$ defined above. In this specific case, N corresponds to `nShares`. In the case of a market featuring `5` stocks, for example, `w` will look like:

```
[0.2, 0.2, 0.2, 0.2, 0.2]
```

It is time now to translate such percentages into actual stocks quantities. The goal is achieved by means af a second `for` loop:

```
for n in range(len(w)):
    w[z] = int(w[z] * budget)
```

This makes so that the `z`-th element of the above `w` vector will be multiplied by the investor's budget (`trader[0]`) and that, to avoid the trading of fractions of stocks, the resulting number will be set equal to its integer part only.

The new `w`, with `nShares = 5` and `trader[0] = 1000`, will then be:

```
[200, 200, 200, 200, 200]
```

The above portfolio, however, corresponds to the ideal one. To actually hold it, each investor needs to modify its previous portfolio. This is exactly what the last loop line is asked to do:

```
we.append(w[z] - oldWeights[nRandom + i][z])
```

Here, the `we` vector is filled with the difference between the desired quantity of the `z`-th stock, `w[z]`, and the current quantity held at the moment, contained in the `oldWeights` list. Notice how, in order to reach the correct investor's list, the index required cannot be `i` alone. This would make the program pick an `oldWeights` element which actually corresponds to a random investor. Adding `nRandom` to `i` makes possible to skip all the undesired random investors, and work with equally weighted investors only.

In the end, `weights` is expanded with the list `we`:

```
weights.append(we)
```

Momentum traders

Momentum traders follow a simple rule: *buy winners, sell losers.*

The strategy consist in monitoring the market for a given amount of time (called *formation period*) and, at the end of such period, ranking the stocks based on their realized returns. The best ones form the *winners* portfolio, the worse ones form the *losers* one. The portfolio to be held, of course, is the former.

As usual, the whole process is contained into a loop of the form:

```
for i in range(nMomentum):
```

However, given their special nature, momentum strategies cannot be applied during the very first cycles of the market. This, indeed, would make the formation period excessively short (if not non-existent at all, at `t = 0`). This reflects into Python code as well, requiring the whole investor's action to be subordinated to a `if` condition of the type:

```
if t > 8 and t % 3 == 0:
```

This ensures an initial formation period equal to `9` market cycles and, also, that the momentum portfolio will be rebalanced every `3` cycles. The second condition, indeed, verifies that the reminder of the ratio `t / 3` exactly equals `0`. Only in that case, momentum investors are allowed to operate.

Before proceeding, it is necessary to write down the code enabling investors to evaluate stocks performances based on the formation period. A dedicated empty array is generated:

```
performances = []
```

Performances, here, consist in the mean log-return of each stock over the past `9` market cycles. Such value is computed for each share as follows:

```
for j in range(nShares):
    a = []
```

Here, `a` is a temporary list that will be filled with the past `9` log-returns of the `j`-th stock. This is accomplished by means of a second `for` loop:

```
for s in range(t-9, t-1):
    a.append(lnReturns[j][s])
```

The object appended to `a` corresponds to the log-return of the `j`-th stock at time `s`, which is contained in the interval `(t-9, t-1)`.

Once `a` is filled, numpy's `mean` function is exploited to compute the vector's mean:

```
m = np.mean(a)
```

The resulting value is the attached to the `performances` list:

```
performances.append(m)
```

The last step consist in sorting the stock's performances by means of the appropriate function:

```
p = sorted(performances, reverse = True)
```

The `reverse` condition ensures that the objects contained inside `p` will be decreasingly sorted, since the default condition is that of an increasing sorting.

It is now possible to compute the quantities to be bought. At first, the usual empty vectors that will contain the weights and the percentages are created:

```
we = []
w = []
```

The required *modus operandi*, now, is the following:

```
for n in range(len(p)):
    if performances[n] != 0 and performances[n] == p[0]
    or performances[n] != 0 and performances[n] == p[1]:
        w.append(1./2)
```

For every element contained in `performances` (i.e. the unsorted list), investors have to verify two conditions: the `n`-th performance must be different from `0` and, also, it must be equal to one of the first two elements of the `p` list (i.e. the first elements of the sorted list, corresponding to the best mean returns over the last `9` market cycles). Only if this is the case, `w` is increased by a positive percentage, depending on the stocks quantities under consideration. In this example, `nShares` is set equal to `5` and the momentum portfolio consists of the two best performing stocks. Hence, the corresponding portfolio weight will be `1./2`.

To conclude, whenever one of the above `if` conditions is not satisfied, the corresponding portfolio share must be set equal to `0`:

```
else:
    w.append(0.)
```

This last loop brings up a problem: whenever the second best mean return equals `0.0`, the `w` list will look something like:

```
[0.0, 0.5, 0.0, 0.0, 0.0]
```

This, of course, is not efficient, since the investor is allocating only half of its portfolio.

To face this situation, it is necessary to exploit a precise `if` condition, checking whether the sum of the portfolio's weights actually equals one:

```
if sum(w) != 1.0:
```

When it is not the case, the program has to scan each `w`'s element:

```
for n in range(len(w)):
```

looking for the only one that corresponds to a positive weight:

```
if w[n] != 0.0:
```

thus to make it equal to one:

```
w[n] = 1.0
```

This ensures a full portfolio allocation even in the case in which the market contains one stock only featuring a positive momentum.

As always, the last step consists in confronting the ideal portfolio composition with the actual one, in order to find the quantities that compensate the difference:

```
for z in range(len(w)):
    w[z] = int(w[z] * trader[0])
    we.append(w[z] - oldWeights[nRandom + nEqually + i][z])
```

Notice how, again, the indexing needs to take into account all the previous investors categories in order to properly work. Hence, `i` needs to be increased by the quantity `nRandom + nEqually`.
To conclude, the usual append process is executed:

```
weights.append(we)
```

Momentum reversal traders

Momentum reversal strategy, as the name suggests, is nothing but the reversed momentum strategy. The stocks ranking is still performed but, this time, the losers are held instead of the winners.

Such similarity reflects also on the code, which differs from the above one in two points only.

First, after the stocks performances are computed, the sorting is reversed:

```
p = sorted(performances)
```

The absence of the `reverse` condition specification ensures a decreasing list creation. At this point, `p[0]` and `p[1]` will correspond to the two worse performances, implying no need to further modifications of the code.

Second, when characterizing the `we` vector, the quantity `nMomentum` has to be considered:

```
we.append(w[z] - oldWeights[nRandom + nEqually +
                            nMomentum + i][z])
```

High book/market traders

This portfolio strategy derives directly from the three-factor model developed by Fama and French, who found how the ratio between a stock's book value and its market value, as well as its capitalization, are able to explain the stock's return behaviour. In particular, returns seem to have a positive correlation with the book-to-market ratio (i.e. a higher ratio implies higher returns).

Taking this into account, the present strategy seeks at creating a portfolio containing only the best performing stocks, from a book-to-market ratio point of view.

First of all, it must be noticed that the strategy cannot be run during the very first market cycle (`t == 0`). Indeed, this would imply computing a book-to-market ratio which is actually impossible to compute, since there exists no market price yet. One could assume it to be equal to the book price, which makes sense, but this would lead to a ratio equal to one for any stock, making the strategy pointless.

This is why the traders characterization is subordinated to the following `if` condition:

```
for i in range(nHigh):
    if t > 0:
```

Speaking about the weights, the first step to compute them is to define the stocks book-to-market values. Hence, after the creation the empty vector:

```
BM = []
```

a `for` loop is applied to each of the shares in the market:

```
for z in range(nShares):
```

Here, the book-to-market value is computed as follows:

```
bm = initialPrices[z]/officialPrices[z][t-1]
```

as the ratio between the `z`-th stock's initial price and its official price at the end of the last market cycle (i.e. at the beginning of the present market cycle).

This value is the appended to the `BM` list:

```
BM.append(bm)
```

Since the goal is to individuate the best performing shares, the above list is sorted, exactly as seen in the momentum case:

```
sortBM = sorted(BM, reverse = True)
```

Again, in order to obtain a decreasing ordering, the `reverse` condition must be explicitly called.

At this point, the portfolio's percentages can be defined, to be collected in the vector `w`:

```
w = []
```

and, after the usual computations, to be collected inside `we`:

```
we = []
```

Each element contained in `BM`:

```
for n in range(len(BM)):
```

is confronted to the elements belonging to `sortBM`. If the `n`-th book-to-market ratio corresponds to one of the best values:

```
if BM[n] == sortBM[0] or BM[n] == sortBM[1]:
```

the corresponding stock is featured with a positive portfolio's share:

```
w.append(1./2)
```

Otherwise, it will not be bought:

```
else:
    w.append(0.)
```

As for the momentum case, notice that the weight to be assigned corresponds to `1./2`: this is so simply because the above example features a total number of shares in the market equal to `5`.

As usual, the last step consists in confronting the old portfolio composition with the new, desired one in order to exactly compute the compensating amounts needed to reach the goal:

```
for z in range(len(w)):
    w[z] = int(w[z] * trader[0])
    we.append(w[z] - oldWeights[nRandon + nEqually +
                                nMomentum + nReversal + i][z])
```

As for the momentum case, notice how the index `i` needs to be increased by the appropriate amount (here, corresponding to `nRandom + nEqually + nMomentum + nReversal`).

To conclude:

```
weights.append(we)
```

Low book/market traders

Similarly to the reversal momentum strategy, the low book-to-market ratio one simply consists in the opposite of the original strategy. In this case, hence, the trader will look for those stocks showing the worst ratio values in the market.

The functioning is exactly equal to the one just presented above, except for the ordering operation. Omitting the `reverse` command, `sortBM` corresponds now to an increasing list:

```
sortBM = sorted(BM)
```

This means that, now, `sortBM[0]` and `sortBM[1]` correspond to the worst book-to-market values, which are exactly what the traders are looking for.

Moreover, remember to add the `nHigh` quantity when computing the weights:

```
we.append(w[z] - oldWeights[nRandon + nEqually + nMomentum
                            + nReversal + nHigh + i][z])
```

Small capitalization traders

In their three-factor model, Fama and French also revealed a second relationship between asset returns and their determinants. They shown, indeed, that there exists a negative correlation between expected returns and capitalization. Small firms, hence, tend to perform better.

The capitalization is an indicator computed as follows:

$$capitalization_{(i,t)} = P_{(i,t)} \cdot N_{(i,t)}$$

where the capitalization of asset i at time t is given by the product between the asset's price ($P_{(i,t)}$) and the number of stocks present in the market at that time ($N_{(i,t)}$).

Moving on to the code, each trader:

```
for i in range(nSmall):
```

first needs to compute and rank stocks capitalizations, which will be stored in the empty vector `Cap`:

```
Cap = []
```

If investors find themselves in the first market cycle:

```
if t == 0:
```

for each share:

```
for j in range(nShares):
```

they will compute a value, `a`, corresponding exactly to the capitalization formula illustrated above:

```
a = initialQuant[j] * initialPrices[j]
```

In this case, `initialQuant[j]` corresponds to $N_{(j,t)}$, while `initialprices[j]` coincides with $P_{(j,t)}$.

Right after, `a` is appended to the `Cap` list:

```
Cap.append(a)
```

The same exact procedure applies in the case of `t > 0`, with one unique difference: the price to be inserted in the formula corresponds now to the stock's official price generated at time `t-1`:

```
if t > 0:
    for j in range(nShares):
        a = initialQuant[j] * officialPrices[j][t-1]
        Cap.append(a)
```

To rank the indicators, `Cap` needs to be sorted:

```
sortCap = sorted(Cap)
```

Notice the absence of the `reverse` parameter, ensuring the creation of an increasing list of values.

Once the process is completed, the usual vectors can be generated:

```
we = []
w = []
```

and filled.

Each stock's capitalization is checked:

```
for n in range(len(Cap)):
```

to verify weather it corresponds to one of the smallest (i.e. to one of the first values of `sortCap`):

```
if Cap[n] == sortCap[0] or Cap[n] == sortCap[1]:
```

Whenever this is the case, the stock is inserted inside the portfolio with the appropriate weight:

```
w.append(1./2)
```

Otherwise, it will not be bought (i.e. its weight will correspond to zero):

```
else:
    w.append(0.)
```

At last, as always, each element of the vector `w` must be multiplied by `budget`, while `we` must be appended with the difference between the target and the old portfolio's compositions:

```
for z in range(len(w)):
    w[z] = int(w[z] * budget)
    we.append(w[z] - oldWeights[nRandom + nEqually +
                                nMomentum + nReversal +
                                nHigh + nLow + i][z])
```

Lastly, `we` can be appended to `weights`:

```
weights.append(we)
```

Big capitalization traders

Exactly as seen for the momentum reversal and for the low book-to-market strategies, the big capitalization one is identical to the previous technique, except for two elements: the `Cap` vector must be sorted following a decreasing order, thus to have the biggest values in the first positions:

```
sortCap = sorted(Cap, reverse = True)
```

while the first index required to individuate the correct old weights has to be increased by the value `nSmall`:

```
we.append(w[z] - oldWeights[nRandom + nEqually + nMomentum +
          nReversal + nHigh + nLow + nSmall + i][z])
```

Genetic traders

Genetic traders, as already seen in section 4.2, individuate an objective function to be optimized and, by means of genetic algorithms, find the portfolio composition that allows for such optimization. Goal functions can be of any type, and to each case corresponds a different Python coding.

Here, only one case is presented, featuring the following evaluation function:

$$f(g) = \frac{E_t(r_{(p,t+1)})}{\sigma^2_{(p,t+1)}}$$

The numerator represents the expected value, in time t, of the portfolio's logarithmic return at time $t+1$. The denominator, instead, corresponds to the portfolio's variance at time $t+1$.

Since to compute such values, the investor needs data coming from the stocks time series of log-returns, it seems appropriate to consider a period of time devoted to data collection. In code terms, for example, this can translate into an `if` condition:

```
for i in range(nGenetic):
    if t > 4:
```

Here, after the usual `for` loop, the individual is given a period of four market cycles before actually starting trading.

As for all the other categories, the first step for genetic traders is to find their ideal portfolio composition. This is where pyevolve operates.

First of all, it is indispensable to define the desired evaluation function:

```
def eval_func(genome):
```

As seen above, the goal is to compute a ratio between a mean and a variance, which of course need to be computed. This is not done here but, instead, the task is accomplished further in the code. This does not obstacle the correct functioning of the genetic processes since, while means and variances are computed at the end of every cycle, starting from `t == 0`, genetic algorithms are run starting from `t == 5`. Hence, for the moment, the explicit computations are omitted and will be shown later. All it is sufficient to know, for the time being, is that the program generates two dedicated vectors, named `means` and `variances`. From the latter, the covariance matrix `cov` is then derived.

Going back to the evaluation function, it is necessary to define all the variables involved in its computation:

```
score = 0.0
ret = 0.0
risk = 0.0
```

where `ret` denotes $E_t(r_{(p,t+1)})$, while `risk` indicates $\sigma^2_{(p,t+1)}$.

Next step, of course, is to compute such values.

Since `ret` corresponds to the expected logarithmic return of the portfolio, it is sufficient to increase the initial value, for a number of time coinciding

with the total number of stocks, by the expected log-return of the `j`-th share (`mean[j][t-1]`) multiplied by the corresponding portfolio fraction, expressed in genetic algorithms terms (`genome[j]`):

```
for j in range(nShares):
    ret += genome[j] * means[j][t-1]
```

according to the formula:

$$E_t(r_{(p,t+1)}) = \sum_{j=1}^{N} \omega_{(j,t)} \mu_{(j,t)}$$

where $\omega_{(j,t)}$ is the stock's weight, while $\mu_{(j,t)}$ is the stock expected log-return computed in time t.

The same logic applies to the computation of `risk`, which is the risk associated to the portfolio. It is computed as:

$$\sigma^2_{(p,t+1)} = \sum_{j=1}^{N} \sum_{k=1}^{N} \omega_{(j,t)} \omega_{(k,t)} \sigma_{([j,k],t)}$$

with $\sigma_{([j,k],t)}$ equal to the covariance between the log-returns of the j-th and the k-th stocks (i.e. the `j`-th element of the `k`-th row of the matrix `cov`).

A double sum, in Python, corresponds to a double `for` loop:

```
for j in range(nShares):
    for k in range(nShares):
            risk += genome[i] * genome [j] * cov[j][k]
```

To obtain the score value `score`, then, it is sufficient to compute the appropriate ratio, given a denominator different from zero:

```
if risk != 0:
    score = ret / risk
```

Finally, to properly end the function definition, a last line is necessary:

```
return score
```

After the evaluation function is defined, pyevolve requires to set a series of parameters. To start, the genome is characterized:

```
genome = G1DList.G1DList(5)
genome.evaluator.set(eval_func)
genome.mutator.set(Mutators.G1DListMutatorRealGaussian)
genome.initializator.set(Initializators.G1DListInitializatorReal)
```

The algorithm must search for a one-dimensional solution of length five (`G1DList(5)`) evaluated according to the above `eval_func`, mutating individuals based on a real Gaussian mutator and operating with real values.

Then, the algorithm's features are specified:

```
ga = GSimpleGA.GSimpleGA(genome)
ga.setGenerations(1000)
pop = ga.getPopulation()
pop.scaleMethod.set(Scaling.SigmaTruncScaling)
ga.setPopulationSize(400)
ga.evolve(freq_stats=250)
```

Here, pyevolve must apply its `GSimpleGA` function on `1000` generations containing `400` individuals each, printing on screen statistics every `250` generations. Third and fourth lines above deal with the sigma truncation scaling, which is indispensable here, since there exists the possibility of negative results that, without such settings, would result in a program error.

At the end of this process, pyevolve writes the optimal solution inside an element called `ga.bestIndividual()`. For sake of simplicity, it is renamed:

```
best = ga.bestIndividual()
```

and, also, the usual two empty vectors are generated:

```
we = []
g = []
```

At this point, it must be noticed how the genetic optimization set above is not constrained with regards to the sum of its values. This means that the resulting portfolio weights do not sum up to one, as they actually should. The problem is easily overcome by means of a simple normalization.

Every value contained in `best`:

```
for j in range(len(best)):
```

is hence divided by the total sum of the values:

```
a = best[j]/sum(best)
```

and appended to `g`:

```
g.append(a)
```

After those operations, `g` sums up to one, i.e.:

```
sum(g) == 1.0
```

resulting in:

```
True
```

What is left to do is to scale the portfolio percentages by the trader's budget:

```
for z in range(len(g)):
    g[z] = int(g[z] * budget)
```

and then fill the list `we` with the differences between the old and the goal composition:

```
we.append(w[z] - oldWeights[nRandom + nEqually + nMomentum +
                            nReversal + nHigh + nLow + nSmall +
                            nBig + i][z])
```

Finally, `we` must be appended to `weights`:

```
weights.append(we)
```

Since the whole code started with an `if` condition, it is necessary to define genetic traders behaviour in market cycles prior to `t == 5`. In such case, all the weights must equal zero, thus to ensure the individuals inaction:

```
else:
    we = []
    for j in range(nShares):
        we.append(0.0)
    weights.append(we)
```

Once all the portfolios have been characterized, it is possible to go back to the trader's general features.

Since the code is going to operate on every individual again, it is necessary to create an appropriate `for` loop:

```
for i in range(len(weights)):
```

The choice here is to iterate the loop over a quantity equal to the length of the `weights` vector, that contains as many portfolio compositions as the number of traders in the market. Hence, it would be equivalent to iterate the loop over a range equal to `nTotal`.

After that, the empty vector deputed to containing all the relevant information is generated:

```
trader = []
```

and, immediately after that, the arbitrary value defined at the very beginning of the code is called:

```
trader.append(budget)
```

and placed in position `0` of the `trader` vector.

As already said above, the reason for the existence of such element is merely technical but, broadly speaking, it can be thought as some sort of trader's budget, helping defining the maximum amount of stocks that each individual can buy during each market cycle.

So far, the operations performed were referring to each single trader as a whole. At this point, it is necessary to operate at a lower level, defining the trader's positions with respect to each single share offered in the market. This, of course, is achieved by means of a `for` loop iterated a number of times equal to `nShares`:

```
for j in range(nShares):
    share = []
```

The empty vector `share` will be devoted to the storage of buy and sell prices, buy and sell quantities and of two binary variables that indicate whether the individual wants to trade or not.

First, in order to be admitted to the market operations, traders need to express, for each share, a buy price as well as a sell price, which will later be collected to create the book. With respect to those variables, not only the random traders act randomly:

```
if t == 0:
    share.append(initialPrices[j]+round(uniform(-5,10),3))
    share.append(initialPrices[j]+round(uniform(-5,10),3))
if t > 0:
    share.append(officialPrices[j][t-1]+round(uniform(-5,10),3))
    share.append(officialPrices[j][t-1]+round(uniform(-5,10),3))
```

Even though the commands are split into two different parts, their functioning is always the same: the previously created **share** vector is filled with two elements. At position **0** the buy price is set, while at position **1** follows the sell price. How those prices are computed is intuitive: first of all, a starting point is needed, which is always represented by the previous day's (**t-1**) official stock price.

Distinguishing between the two cases, **t == 0** and **t > 1**, is hence necessary because at the beginning of the first market cycle no previous time exists, and the official prices are substituted by the values contained inside the **initialPrices** vector.

Such value, anyway, is then modified by each agent in order to ensure some market dynamics to emerge. The function **uniform()** draws an observation from the specified range, according to a Uniform distribution. This value is then rounded the third decimal value, just for sake of simplicity in further data analysis. Notice that the lower bound of the range is, in absolute value, lower than the upper one. This is necessary in order to avoid prices from falling steadily, which would happen with equal values on both bounds because of the nature of the market book, which is going to be analyzed soon.

Through the **append** function, then, those two newly generated values are attached to the **share** vector which, at this point, may look something like that:

```
[113.735, 109.147]
```

Next step is to associate buying and selling quantities to each stock:

```
if weights[i][j] > 0:
    share.append(weights[i][j])
else:
    share.append(0)

if weights[i][j] < 0:
    share.append(abs(weights[i][j]))
else:
    share.append(0)
```

There is no really need to perform any computations here, it is sufficient to rely on what was already calculated when the **weights** vector was created and filled. All is left to do is to apply the appropriate `if` and `else` conditions: whenever the portfolio quantity associated to the **j**-th stock of the **i**-th individual is positive, the buying quantity is set equal to the portfolio's one and appended to **share**, while the selling quantity is set equal to **0**. If, instead, the portfolio quantity is negative, the whole operation is reversed: a **0** is appended, followed by the absolute value of the negative portfolio

quantity. This is because, when creating the books, negative quantities are unwanted.

In the two cases, `share` would look this way:

```
[113.735, 109.147, 124, 0]
[113.735, 109.147, 0, 124]
```

Last, fundamental element to be created is the binary variable denoting if the investor is willing to buy, sell, or neither of the two:

```
if share[2] != 0:
    share.append(1)
else:
    share.append(0)

if share[3] != 0:
    share.append(1)
else:
    share.append(0)
```

The reasoning is simple: if the buying quantity (`share[2]`) is positive, the trader wants to buy and the associated binary variable takes value `1`, `0` otherwise; if the selling quantity (`share[3]`) is positive, the trader wants to sell and the associated binary variable takes value `1`, `0` otherwise.

The share vector now looks this way:

```
[113.735, 109.147, 124, 0, 1, 0]
[113.735, 109.147, 0, 124, 0 ,1]
```

All such passages are iterated for all the stocks, and the resulting vector is each time appended to the `trader` array:

```
trader.append(share)
```

At this point, any operation aimed at characterizing random traders is completed, so the last thing to do is to append the `trader` vector to the `traders` one, which contains them all:

```
traders.append(trader)
```

Since there is a risk of confusion, I want to explicitly recap what is the content and the structure of each main vector at this point:

With `5` traders, `traders`:

```
[[trader 1 vector], [trader 2 vector], [trader 3 vector],
 [trader 4 vector], [trader 5 vector]]
```

With `5` shares, each **`trader`** vector:

```
[budget, [share 1 vector], [share 2 vector],
 [share 3 vector], [share 4 vector], [share 5 vector]]
```

Finally, each **`share`** vector:

```
[buy price, sell price, buy quantity,
 sell quantity, buy yes/no, sell yes/no]
```

5.3 Market maker

To the market maker is devoted a separate piece of code because of its special functions. Its action is, of course, devoted to ensure the market's liquidity by buying and selling great amounts of stocks, thus to allow all the other traders to constantly have a counterpart for the desired transactions. At the same time, however, it is the subject that, during the very first market cycle, has the duty of presenting to the world the entire amount of stocks that the (ideal) firms are willing to sell.

As for the traders, the first step consists in generating the dedicated vector, and appending the budget to it:

```
marketMaker = []
marketMaker.append(mmBudget)
```

After that, exactly as before, it is necessary to create the **`share`** vectors, one for each stock to be traded:

```
for i in range(nShares):
    share = []
```

The difference with the previous case is made evident in the following lines of code.

At time zero, indeed, the market maker has to offer to the market extremely precise quantities of stocks, at precise prices, corresponding to the values contained in the vectors **`initialQuant`** and **`initialPrices`**, created at the beginning of the program:

```
if t == 0:
    share.append(0.0)
    share.append(initialPrices[i])
    share.append(0.0)
    share.append(initialQuant[i])
```

The first and the third elements appended to `share` represent respectively the buying price and the buying quantity for the `i`-th stock. However, because of the special role it has during the first cycle, the market maker will only sell stocks. Notice how this cannot influence market's liquidity, since at time `t == 0` the other traders active in the market have an empty portfolio and, also, will not take short positions by construction. This implies that only one subject will sell (the market maker), and that investors that are buying cannot also sell.

From `t > 0` on, however, the situation changes and becomes more general. In this case, the market maker will form its buying and selling prices as usual:

```
if t > 0:
    share.append(officialPrices[i][t-1] +
                 round(uniform(0, 10), 3))
    share.append(officialPrices[i][t-1] +
                 round(uniform(0, 0), 3))
```

while the buying and selling quantities need to respect a simple rule. The market maker, indeed, cannot hold an amount of stocks higher than the initial one (`initialQuant`) and, also, cannot sell more stocks than the ones present in its portfolio (`portfolios[nRandom + nEqually + nMomentum + nReversal + nHigh + nLow + nSmall + nBig]`).

This results in the following code:

```
share.append(initialQuant[i] - portfolios[nRandom + nEqually +
             nMomentum + nReversal + nHigh + nLow + nSmall +
             nBig][i])
share.append(portfolios[nRandom + nEqually + nMomentum +
             nReversal + nHigh + nLow + nSmall + nBig][i])
```

Then, the binary variables are generated as usual:

```
if share[2] != 0:
    share.append(1)
else:
    share.append(0)

if share[3] != 0:
    share.append(1)
else:
    share.append(0)
```

and the market `share` vector is appended to the `marketMaker`:

```
marketMaker.append(share)
```

To conclude, it is necessary to define the market maker's portfolio. The procedure to create it is not different to the one applied to all the other traders, the only difference is that, at time zero, its portfolio will contain the entire amount of stocks available (i.e. `initialQuant`):

```
if t == 0:
    ptf = []
    for j in range(nShares):
        ptf.append(initialQuant[j])
    portfolios.append(ptf)
```

When everything is concluded, it is possible to add the market maker to the `traders` list:

```
traders.append(marketMaker)
```

5.4 Books

At this point, all the counterparts that will participate to the market operations have been defined and that means that, for the current market cycle, the information about all the buy and sell prices are available and ready to be manipulated.

The functioning of such books is quite simple. All the different prices are collected into a unique list, which is then sorted based on its nature: the buying offers follow a decreasing order, while selling offers are sorted according to an increasing order.

To achieve that same result with coding, three passages are needed. In the first one, all the relevant data are collected and divided by type.

First of all, the vector that will contain such information needs to be generated:

```
buyLists = []
```

Now, two `for` loops are needed to accomplish the task.The scope, indeed, is to collect each share's price (first loop) for each individual (second loop). Hence:

```
for i in range(1, nShares + 1):
    buy = []
```

allows to sort shares contained in each `trader` vector. The reason for a range shifted by `1`, `(1, nShares + 1)`, is the presence, in position `trader[0]`, of what has been defined as the trader's budget . With this shifting, then, the loop will analyze elements going from `trader[1]` to `trader[nShares]`, which are nothing but the desired `share` vectors.

Moreover, the `buy` empty array is created, to be filled soon. It represents the buy book for the `i`-th share.

What is left to do, at this point, is to sort traders:

```
for j in range(len(traders)):
    a = [traders[j][i][0], traders[j][i][2]]
```

where the length of the `traders` list coincides with the total number of individuals operating in the market.

For each one of them, a vector (named `a`) is created to contain both the trader's buying price (`traders[j][j][0]`) and the related quantities (`traders[j][i][2]`).

To clarify, take `traders[j][i][0]`: such notation, going backwards in considering the three parenthesis, indicates the first element of the `i`-th `share` vector for the `j`-th investor.

Based on whether the individual is buying or selling, the considered quantity may be equal to `0`. This is why the following `if` condition is used:

```
if traders[j][i][2] != 0:
    a.append(1)
else:
    a.append(0)
```

If the individual is buying (i.e. the quantity is not equal to zero, `!= 0`) the list `a` is increased buy a `1`, if the individual is selling a `0` is appended. This is nothing but the same binary variable that is present in each `trader` vector.

After the whole process, `a` takes one of those two forms:

```
[traders[j][j][0], traders[j][i][2], 1]
[traders[j][j][0], traders[j][i][2], 0]
```

and is appended to the previously created list, `buy`:

```
buy.append(a)
```

Lastly, `buy` is appended itself to the `buyLists` array, which contains all the market's buy books:

```
buyLists.append(buy)
```

The same procedure, of course, is applied to the sell prices:

```
sellLists = []
for i in range(1, nShares + 1):
    sell = []
        for j in range(len(traders)):
            a = [traders[j][i][1], traders[j][i][3]]
            if traders[j][i][3] != 0:
                a.append(1)
            else:
                a.append(0)
            sell.append(a)
        sellLists.append(sell)
```

All such lists, however, are sorted buy the order followed for the creation of traders, which is almost sure will not coincide to the desired order.

By means of another `for` loop, hence, the problem is faced. The need is to extrapolate each one of the lists contained inside `buyLists` and `sellLists` and give them the appropriate ordering. It follows that the code will be:

```
buy = []
for i in range(len(buyLists)):
```

The sorting operations required, then, are actually two: the first one orders the elements according to a decreasing prices fashion:

```
b = sorted(buyLists[i], reverse = True)
```

The second one, instead, avoids that investors offering a high buy price, but not actually being interested in buying the stock (i.e. with a binary variable equal to `0`), are placed at the top of the list by sorting again the elements based on their binary value:

```
b = sorted(b, key = itemgetter(2), reverse = True)
```

Hence, while keeping the decreasing price fashion, bids associated to a `0` binary value are positioned at the end of the book, no matter what the offered price is.

Notice how, in the second case, it is necessary to explicitly indicate the key for the sorting operation by mean of the `itemgetter` function: the `sorted` function, indeed, automatically sorts lists based on their first element. Also, it is necessary to set the value `reverse` as `True`, since the default sorting operation would create an increasing price list.

At last, the sorted list `b` is appended to the `buy` vector, which is going to store, hence, the actual books.

The same procedure applies to the sell case:

```
sell = []
for i in range(len(sellLists)):
    s = sorted(sellLists[i])
    s = sorted(s, key = itemgetter(2), reverse = True)
    sell.append(s)
```

Notice how, in this case, the first sorting does not need any **reverse** variable specification, given that the desired list is an increasing price one.

5.5 Match demand and supply

The previous coding allows to have, for each stock traded, a decreasing buy list together with an increasing sell list. They now have to be matched. The principle according to which a buyer is associated to a seller is simple: the system picks the first element of the buy book (i.e. the highest price offered) and checks the first element of the sell book (i.e. the lowest price asked); if the former is greater than or equal to the latter, the transaction is executed at the lowest among the two prices. The same applies to the quantities exchanged, which again correspond to the lowest among the two considered. This is trivial since, for example, it would not be possible for a seller to sell more stocks than the ones it owns.

Speaking about the code, such matching operations have to be performed for each one of the shares that can be found in the market. This, as already seen many times before, translates into a `for` loop that iterates itself a number of times equal to the number of stocks, **nShares**:

```
for i in range(nShares):
```

The first commands executed inside the loop, however, are about data collection, that will turn out to be useful in a second moment:

```
exe = []
quant = []
```

Indeed, two empty arrays, **exe** and **quant**, are generated. They will later be filled with execution prices and execution quantities respectively.

It is time now to actually match demand and supply. The theoretical conditions for a match to happen have already been explained above, so it is sufficient to translate them into Python code:

```
while buy[i][0][0] >= sell[i][0][0] and buy[i][0][1] != 0
and sell[i][0][1] != 0:
```

The first one checks whether the first element of the buy book (`buy[i][0][0]`) is greater than or equal to the first element of the sell book (`sell[i][0][0]`). To exactly accomplish the required task, notice how a `while` loop is exploited: it ensures that the matching between bid and ask is continuously performed until buying offers are too low for sellers.

Second condition ensures that the quantity to be bought (`buy[i][0][1]`) is different from zero, while third condition does the same with reference to the selling counterpart (`sell[i][0][1]`).

Notice how the last two conditions would be perfectly equivalent to the following:

```
buy[i][0][2] != 0 and sell[i][0][2] != 0
```

since `buy[i][0][2]` and `sell[i][0][2]` denote the binary variable which takes value `0` when the trader is out of the market (i.e. wants to buy or sell an amount of stocks equal to `0`).

As long as the above conditions are satisfied, a transaction takes place, translating into a series of operations to be performed by the system.

First of all, an execution price is set:

```
exePrices[i].append(sell[i][0][0])
```

As mentioned above, it always corresponds to the lower among the buying and selling price. Since, by market construction, the former always needs to be greater than the latter, it follows that the execution prices always corresponds to the selling one. That is why `sell[i][0][0]` is appended to `exePrices[i]`, which is the `i`-th list contained inside the `exePrices` vector, corresponding to the `i`-th stock.

The same operation is then performed with respect to the empty array `exe`:

```
exe.append(sell[i][0][0])
```

This is because `exe` is a temporary element that will be used in further computations and then will be reset, so it is convenient to distinguish it from `exePrices`, which is instead a permanent list.

It is then the turn to individuate the execution quantity, which follows the same rules seen for the prices:

```
exeQuant[i].append(min(buy[i][0][1], sell[i][0][1]))
```

The difference, here, is that it is no more obvious which one among the buying (`buy[i][0][1]`) and the selling quantity (`sell[i][0][1]`) is the

lowest, hence it is required to explicitly call the `min` function. The resulting value is then appended to the `i`-th element of the `exeQuant` list.

As done above, then, the same operation is performed with respect to the temporary vector `quant`:

```
quant.append(min(buy[i][0][1], sell[i][0][1]))
```

Once that the trading operation has been individuated, it is not sufficient to store the execution price and quantity . The exchange, indeed, casts consequences on traders portfolios as well as on the market books themselves, which must be updated with the new values and re-sorted.

The portfolio manipulation requires another `for` loop, and the reason why will be made clear in a moment, after having presented the code:

```
for j in range(len(traders)):
    if buy[i][0][0] == traders[j][i+1][0]:
        portfolios[j][i] += (min(buy[i][0][1], sell[i][0][1]))
```

What is happening here is the `traders` vector being scanned by means of the loop, checking the buying price associated to each investor (`traders[j][i+1][0]`). Please notice that to correctly individuate the `i`-th share, the notation `[i+1]` is needed, since the first element of each `trader` vector corresponds to what has been above defined as budget.

Once the loop individuates the buying price which corresponds exactly to the buying price which was at the top of the book (`buy[i][0][0]`), and by means of an `if` condition, the `i`-th element of the portfolio associated to the `j`-th individual is reached, to be modified according to the execution quantity.

Of course, with reference to the same execution quantity, the same is done for the selling investor:

```
if sell[i][0][0] == traders[j][i+1][1]:
    portfolios[j][i] -= (min(buy[i][0][1], sell[i][0][1]))
```

What also changed, however, is the quantity that the two individuals are still willing to buy or sell, that is the amount entering the book for the current market cycle, which hence need to be modified:

```
buy[i][0][1] -=min(buy[i][0][1], sell[i][0][1])
```

At first, the quantity that the first buying trader is seeking in the market (`buy[i][0][1]`) is reduced by the amount of stocks that have just being bought in the above matching operation (i.e. the execution quantity). After that, the possibility is that the individual is satisfied, meaning that the residual desired amount of shares equals zero:

```
if buy[i][0][1] <= 0:
    buy[i][0][2] = 0
```

In such a case, the related binary variable (`buy[i][0][2]`) is switched to a value of `0` by means of an `if` condition.

The same procedure applies to the selling counterpart:

```
sell[i][0][1] -= min((buy[i][0][1]), sell[i][0][1])
if sell[i][0][1] <= 0:
    sell[i][0][2] = 0
```

At this point, it must be stressed out that Python features a very useful functionality. Whenever a list `b` is created from the sorting of the original list `a`, modifying a value belonging to `b` automatically means modifying the corresponding unsorted values of `a`. This is why, in the present case, it is sufficient to modify the sorted book in order to update the original unsorted `traders` list.

The last passage to be performed is that of re-sorting the books in order to account for the new values coming from the concluded transaction.

The procedure is identical to the one adopted earlier:

```
b = sorted(buy[i], reverse = True)
b = sorted(b, key = itemgetter(2), reverse = True)
```

A new vector `b` is created, containing the newly sorted book. Remember, the first sorting refers to the buying price, the second one refers to the binary variable in position `2`.

To conclude, the old `i`-th book is substituted by the new one:

```
buy[i] = b
```

and the same procedure is applied to the sellers:

```
s = sorted(sell[i])
s = sorted(s, key = itemgetter(2), reverse = True)
sell[i] = s
```

5.6 Official price

In order to define their buying and selling prices at the beginning of the next market cycle, traders need to know what is the official price resulting from the current cycle. This is exactly what is going be computed by means of

the following code.

First of all, recall the formula that allows to compute the official market price:

$$\frac{\sum_{i=1}^{N} exeP_{i,t} \cdot exeQ_{i,t}}{\sum_{i=1}^{N} exeQ_{i,t}}$$

where N denotes the number of transactions concluded for stock i at time t, $exeP_{i,t}$ is the i-th execution price at time t and $exeQ_{i,t}$ the i-th execution quantity at time t.

To compute this in Python, it is first of all necessary to create two variables, one for the numerator (`totalProd`) and one for the denominator (`totalWeights`), that will make the computation of the ratio more intuitive:

```
totalWeights = 0
totalProd = 0
```

In addition to this, to allow the calculation of the summation present in the denominator, a third variable is needed:

```
prod = 0
```

Now the actually computation can start, but only if there has actually been any exchange:

```
if quant:
```

This simple condition checks whether the `quant` list, containing current execution quantities, is not empty. If it is the case, the calculations can start.

In code language, a summation corresponds to a `for` loop, to be iterated a number of times equal to the number of correctly executed operations:

```
for j in range(len(exe)):
```

For each one of them, the denominator is increased by the `j`-th quantity:

```
totalWeights += quant[j]
```

the $exeP_{i,t} \cdot exeQ_{i,t}$ product is computed:

```
prod = exe[j] * quant[j]
```

and it is used to increase the total value of the numerator:

```
totalProd += prod
```

All the needed elements are now at disposal, and the total ratio (i.e. the official price itself) can be computed as:

```
offPrice = totalProd / totalWeights
```

to be then appended to the list inside which all the official prices are stored:

```
officialPrices[i].append(offPrice)
```

In particular, `offPriced` is destined to the `i`-th sub-list belonging to `officialPrices`, the one corresponding to the `i`-th stock.

If the above `if` condition is not satisfied (i.e. no transaction took place during the present market cycle for the `i`-th stock) the situation can be of two types.

In the first case, no transaction happened at time `t` but old official prices were already computed (i.e. `officialPrices[i]` is not empty):

```
else:
    if officialPrices[i]:
```

In such a case, the present official prices is simply set equal to the one resulting from the previous cycle:

```
offPrice = officialPrices[i][t-1]
```

and is the appended to `officialPrices[i]`, exactly as done above:

```
officialPrices[i].append(offPrice)
```

If, instead, no transaction took place during the current cycle and, also, no official price was ever computed for the `i`-th share (i.e. `officialPrices[i]` is empty):

```
if not officialPrices[i]:
```

then today's official price is set equal to the corresponding initial price, generated at `t == 0` and stored in the dedicated list:

```
offPrice = initialPrices[i]
officialPrices[i].append(offPrice)
```

Of course, as always, the official price is finally appended to `officialprices[i]`.

5.7 Relevant indicators

With the re-sorting of the books and the computation of the official price, all the operations strictly related to the market functioning have been defined, and the program could be run the way it is. The focus of this work, however, is on portfolio strategies and, in particular, on their return and risk. This is why a series of computations has to be added at the end of the code, in order to provide all the relevant data that will serve the ultimate purpose of the thesis.

The starting point for any analysis is the computation of stock returns. As already explained in section 4.2, to the end of working on relative instead of absolute price changes, returns are treated in logarithmic form, according to the following formula:

$$r_{i,t} = \frac{P_{i,t}}{P_{i,t-1}}$$

where $r_{i,t}$ stands for the return of stock i at time t and, in the present context, P_i represents the official price of stock i at different times.

As it is evident, in order to compute a return, then, the official price of previous market cycle is needed. It follows that the formula can be applied starting from the second cycle. This situation translates into an `if` condition:

```
if t > 0:
```

If it is satisfied, it is possible to compute the `i`-th stock's return by simply applying the above formula:

```
r = officialPrices[i][t] / officialPrices[i][t-1]
```

Now, obtaining the logarithmic return is straightforward, exploiting the `numpy`'s `log()` function:

```
lnr = np.log(r)
```

To keep track of each of the returns, for each of the market's stocks, they are stored inside the `lnReturns` list, created at the very beginning of the code:

```
lnReturns[i].append(lnr)
```

where each `lnReturns[i]` vector is dedicated to a different share.

If, instead, the market is in its first cycle, the log-return is simply set equal to `0` and stored as before:

```
else:
    lnr = 0
    lnReturns[i].append(lnr)
```

The two basic manipulations of logarithmic returns, of course, are the computations of the time series' mean and variance.

The two can be very easily obtained avoiding the application of the complete statistical formula, by means of two `numpy`'s function.

For the mean:

```
mu = np.mean(lnReturns[i])
```

to be applied to the whole `lnReturns[i]` array and, for the variance:

```
var = np.var(lnReturns[i])
```

The two resulting quantities are then appended to the `means` and the `variances` vector respectively:

```
means[i].append(mu)
variances[i].append(var)
```

so that they can be further manipulated and analyzed.

In order to compute portfolios' risks, however, knowing stocks returns variances is not enough: the complete covariance matrix is needed.

Once again, `numpy` comes in help with a dedicated function, `cov()`, that takes as an argument a matrix. In this particular case, such matrix is the `nShares` x `t` one, filled with the log-returns of the stocks. With `5` shares in the market:

```
mat = np.matrix([lnReturns[0], lnReturns[1], lnReturns[2],
lnReturns[3], lnReturns[4]])
```

that, as explained above, becomes:

```
cov = np.cov(mat)
```

5.8 Portfolios performances

At this point, all the necessary elements are set up and it is possible to actually compute portfolios' performances.

The first step is to calculate the actual composition of each portfolio, since the market dynamics do not ensure that the traders' targets are always satisfied.

The formula to be applied is intuitive:

$$\omega_{i,t} = \frac{quant_{i,t}}{\sum_{i=1}^{N} quant_{i,t}}$$

Here, $\omega_{i,t}$ denotes the percentage of the i-th stock at time t with respect to the total portfolio, while $quant_{i,t}$ is the quantity of the i-th stock present in the portfolio at time t, expressed in number of stocks.

Moving on to the code, as usual, an empty list devoted to the collection of the data that are going to be generated is created:

```
percentages = []
```

In the beginning, it is necessary to be sure that the denominator of the equation above is different from `0`. Only in that case, it is possible to actually compute `\omega_{i, t}`:

```
if sum(portfolios[j]) != 0:
```

If this is the case, then the percentages computations need to be performed for each portfolio present in the market. This means to perform a `for` loop iterated over the number of portfolios:

```
for j in range(len(portfolios)):
```

At first, the denominator is computed as the sum of the `j`-th vector contained in the `portfolios` list:

```
a = sum(portfolios[j])
```

and, after that, there is the need to create an empty array inside which the actually percentages will be collocated:

```
b = []
```

To fill `b` with data, then, a second `for` loop is required:

```
for k in range(len(portfolios[j])):
    b.append(portfolios[j][k]/a)
```

For each element contained inside the `j`-th portfolio, `b` is increased by an element computed exactly as the above $\omega_{i,t}$: the quantity of the `k`-th stock (`portfolio[j][k]`) is divided by the total portfolio quantity (`a`).

The last task consists in appending `b` to the `percentages` list:

```
percentages.append(b)
```

which, after all the computations are finished, takes such a form (in the case of `5` traders):

```
[[b 0], [b 1], [b 2], [b 3], [b 4]]
```

and, with `5` stocks, each `b` is:

```
[omega 0, omega 1, omega 2, omega 3, omega 4]
```

If, instead, the formula denominator equals `0`:

```
else:
```

the choice is to apply the same process to create a portfolio composition featuring only `0.0` percentages:

```
for k in range(len(portfolios[j])):
    b.append(0.0)
percentages.append(b)
```

Moving on to portfolio's returns, it must be noticed that two different types of such indicator can be calculated: the first one corresponds to the realized return obtained by holding the portfolio from time $t-1$ to time t; the second one is the return expected by holding the portfolio from time t to time $t+1$. They are completely different objects and, hence, require to distinct commands.

The former can be formalized by means of the following equation:

$$ln(r_{p,t}) = \sum_{i=1}^{N} \omega_{i,t-1} ln(r_{i,t})$$

with $ln(r_{p,t})$ being the log-return of portfolio p held from $t-1$ to t, $\omega_{i,t-1}$ being the percentage of the i-th stock in portfolio p at time $t-1$ and $ln(r_{i,t})$ being the log-return of the i-th stock in portfolio p and realized over the period $(t-1, t)$.

From the formula it is evident how, to compute such return, the existence of a portfolio at time $t-1$ is crucial. This is why it can be calculated during the first market cycle (i.e. when `t == 0`). An `if` condition can easily handle this:

```
if t > 0:
```

If it is satisfied, the program is asked to create an empty list:

```
ptfReturns = []
```

and to perform a for loop that makes it possible to compute returns for each one of the portfolios:

```
for j in range(len(oldPercentages)):
```

It can be noticed the presence of an element which has not been created already: `oldPercentages`. This is not a problem since, as mentioned above, the loop runs only when `t > 0`, and the `oldpercentages` vector will be created at the end of the first market cycle, in `t == 0`.

After that, the variable associated to the realized return is created:

```
ret = 0
```

and a second for loop is called, this time to express, in code language, the summation seen in the above formula:

```
for k in range(nShares):
    ret += oldPercentages[j][k] * lnReturns[k][t]
```

For any additional `k`-th share, the return is increased by a quantity equal to the product between the share's percentage at cycle $t-1$ (`oldPercentages[j][k]`) and the share's return at cycle t (`lnReturns[k][t]`).

Lastly, the resulting amount is appended to the previously created vector, `ptfReturns`:

```
ptfReturns.append(ret)
```

The second type of return that can be computed is, as said, the expected one. It is expressed by the formula:

$$E_t[ln(r_{p,t+1})] = \sum_{i=1}^{N} \omega_{i,t} E_t[ln(r_{i,t+1})] = \sum_{i=1}^{N} \omega_{i,t} \mu_{i,t}$$

$E_t[ln(r_{p,t+1})]$ corresponds to the return of portfolio p to be realized at time $t+1$, expected in time t. $E_t[ln(r_{i,t+1})$ follows the same logic, applied to the i-th stock, and it corresponds to the time series mean in time t, $\mu_{i,t}$.

As it is evident, the formula is very similar to the one used in the realized return case. This means that also the code will have a high number of similarities. The only thing that must be done, actually, is to change two variables: the percentages to be considered now are the present ones (`percentages`) and the realized returns have to be replaced by their expected values (`means[k][t]`). The overall result is the following:

```
ptfExpReturns = []
for j in range(len(percentages)):
    ret = 0
    for k in range(nShares):
        ret += percentages[j][k] * means[k][t]
    ptfExpReturns.append(ret)
```

Please notice that the usage of `percentages` allows now to perform the computation for any market cycle, `t == 0` included. Hence, no `if` condition is now needed.

At last, since the trade-off between return and risk cannot be neglected, the portfolio's return variance is computed. The related formula is the following:

$$\sigma_{p,t}^2 = \sum_{i=1}^{N} \sum_{j=1}^{N} \omega_i \omega_j \sigma_{(i,j),t}$$

Here, $\sigma_{p,t}^2$ is the variance of portfolio p at time t, while $\sigma_{(i,j),t}$ denotes the covariance, at time t, of the returns of stocks i and j. It is equivalent to the (i, j)-th element of the variance covariance matrix. Of course, with $i = j$, it follows that $\sigma_{(i,j),t} = \sigma_{i,t}^2$.

In the program, the first step is to create and empty array for the variances to be stored in it:

```
ptfRisks = []
```

Then, once again, the computations have to be run for all the investors, hence another `for` loop is required:

```
for j in range(len(percentages)):
```

The variable for the portfolio's variance is now created and set, for the moment, equal to `0`:

```
risk = 0
```

Now, the task is to execute the double sum present in the formula. As already seen, this is equivalent to perform a double `for` loop, executed a number of times corresponding to the number of shares in the portfolio:

```
for k in range(nShares):
    for h in range(nShares):
```

The variance (`risk`) is then increased by the quantity represented by the product of the two stocks weights (`percentages[j][k]` and `percentages[j][h]`) times the covariance between the two returns (i.e. the `k`-th element of the `h`-th column of the covariance matrix, `cov[k][h]`):

```
risk += percentages[j][k] * percentages[j][h] * cov[k][h]
```

The variance is hence calculated and has to be stored inside the dedicated vector:

```
ptfRisks.append(risk)
```

To very last market operations consist in the creation of two lists marked as `old`. Their role, as seen above, is to represent values referring to market cycle `t-1` during the run of market cycle `t`. This is why they are generated at the very end of the code, in order to avoid any overwriting that would cause the loss of the data:

```
oldWeights = portfolios
oldPercentages = percentages
```

This allows to transfer current values, stored inside `portfolios` and `percentages`, to the following period without fearing to execute the above commands that erase their objects and make them empty again.

This operation concludes the market cycle and makes the very first `for` loop restart.

To see the full code, go to Appendix C.

Chapter 6

Market simulations

This chapter presents the results obtained after a series of runs of my artificial stock market.

When analyzing portfolios data, a particular attention must be paid to the values taken under consideration. Given their nature, indeed, stocks present ambiguous features. On one hand, one of the first thing people may want to analyze is their return; on the other hand, risk cannot be neglected. A high expected return may seem appealing and may be considered as a universally good characteristic a stock should present but, as it is well known, it also implies a high risk. Portfolios, being nothing but linear combination of stocks, are no different.

This is why, when looking at the simulations results, it is important not to focus solely on the return or on the risk obtained by a given portfolio. On the contrary, the object of interest should derive directly from the evaluation function used in the genetic algorithms. Such function can take any imaginable form but, in the following analysis, one specific version of it is used. The formula is quite simple:

$$f(g) = \frac{E(r_P)}{\sigma_P^2}$$

yet it is very powerful. Maximizing the ratio between the portfolio's expected return, $E(r_P)$, and its variance, σ_P^2, means making the algorithm look for the portfolio's composition that can give the highest return per unit of risk.

Since this is the goal of the genetic trader, it would make little sense to evaluate its performance based on portfolios returns, or variances, alone. They surely are important features to be considered but, in this case, they cannot be used as a performance benchmark. What truly determines the goodness of the genetic algorithm strategy, however, is the evaluation of its return-over-risk ratio.

Going back to the evaluation function, the choice fell on the above specification because not only it resembles the usual optimization problem faced in financial theory but, also, because it can be easily modified in order to meet specific preferences or needs without loosing its effectiveness. In its basic form, indeed, the function simply forces the algorithm to find the best ratio, regardless of the numerator and denominator values. If, instead, a trader wants to find the portfolio composition ensuring the highest expected return (the lowest variance) given a maximum (minimum) level of risk (of return), all it is sufficient to do is to add the appropriate constraint to the maximization problem.

Please notice that the possible combinations to set up the simulations are extremely large, because of the high number of variables involved. There are general variables such as the number of traders in the market, the number of market cycles to be observed, the traders budget, the initial stock prices and the magnitude of their variations over the cycles; specific variables such as the formation and the holding periods for momentum and momentum reversal traders, the number of best stocks to be bought by momentum and momentum reversal investors, as well as by high and low book-to-market and by big and small cap ones; lastly, there is a whole series of genetic algorithm parameters that is possible to manipulate: the number of generations and the size of the population, the type of crossover as well as the proper mutator, the selection method and the possible maximum and minimum parameters.

Also, in order to have acceptable results from the algorithms, it is not possible to choose a number of operations that is too small, since this would mean giving the evolution not enough time to operate and, in conclusion, would lead to poor performances. This, however, requires hours of work for each program run.

This considered, and given the limited possibilities in terms of both time and computational power, the cases considered in this chapter can be nothing but limited. I chose to fix a set of variables, hence to focus on a sub-group of them which, instead, have been manipulated.

The fixed conditions, common to all the runs, are:

(i) The number of traders is fixed and equal to `5` for each category. The only exception is about the genetic trader investor, which is always single;

(ii) Every trader features the same budget, equal to `1000`. The market maker's budget, instead, is variable;

(iii) The number of shares to be traded is fixed and equal to `5`;

(iv) The formation period chosen for momentum and momentum reversal strategies is fixed and equal to `15` market cycles. The holding period is also fixed and equal to `5` market cycles;

(v) The number of best stocks to be bought is fixed and equal to `2`. This applies to the following strategies: momentum, momentum reversal, high book/market, low book/market, small cap, big cap;

(vi) The number of market cycles to be observed is fixed and equal to `100`;

(vii) The genetic trader always enters the market after `50` cycles, thus to have a quite solid data base to compute stocks expected returns;

(viii) As pointed out above, the evaluation function is fixed and equal to `ret/risk`. Moreover, possible genetic solutions are generated from the range `(0.0, 1.0)`;

(ix) The population size is fixed and equal to `2000` individuals, while the number of generations is fixed and equal to `600`.

6.1 First run: small variance of prices

The first run is characterized by a small variance of stocks prices, since the sell prices asked by every trader can be modified by an amount contained in the range `(-1.0, 1.0)`. Notice also that the range is centered in zero, thus to have no influence on a possible increasing or decreasing behavior of stock prices.

Genetic algorithms exploit a real range mutator, a two-point crossover and a roulette wheel selector.

Before showing the numerical results, it may be useful to clarify some notation. In all the tables presented in this chapter, each row contains the data referring to a specific category of traders. Th corresponding notation is the following:

R1 = first random trader
R2 = second random trader
R3 = third random trader
R4 = fourth random trader
R5 = fifth random trader
EW = equally weighted trader
Mo = momentum trader
MoR = momentum reversal trader
Hi = High book/market trader
Lo = low book/market trader
SC = small cap trader
BC = big cap trader
Ge = genetic trader

MM = market maker

First of all, table 6.1 contains the data referring to the evaluation function, i.e. the expected and realized ratios between mean portfolio return and its variance.

	expRatios	ratios	expRatiosDiff	%	ratiosDiff	%
R1	-46.2	2.65	56.93	-530.49 %	-30.86	109.37 %
R2	-44.75	-6.03	55.48	-516.99 %	-22.19	78.62 %
R3	-41.86	-13.57	52.6	-490.11 %	-14.65	51.93 %
R4	-46.45	-0.99	57.18	-532.83 %	-27.23	96.48 %
R5	-44.21	35.01	54.94	-512.0 %	-63.23	224.08 %
EW	-69.47	-11.76	80.2	-747.34 %	-16.46	58.31 %
Mo	-22.65	24.32	33.38	-311.04 %	-52.54	186.16 %
MoR	-46.4	2.85	57.13	-532.41 %	-31.07	110.1 %
Hi	-54.6	-16.22	65.33	-608.82 %	-12.0	42.54 %
Lo	14.62	-13.58	-3.89	36.26 %	-14.64	51.89 %
SC	-33.64	9.54	44.37	-413.45 %	-37.76	133.79 %
BC	-45.77	-14.23	56.5	-526.5 %	-13.99	49.57 %
Ge	10.73	-28.22	0.0	0.0 %	0.0	0.0 %
MM	-67.95	-8.67	78.68	-733.18 %	-19.55	69.27 %

Table 6.1: Expected and realized ratios

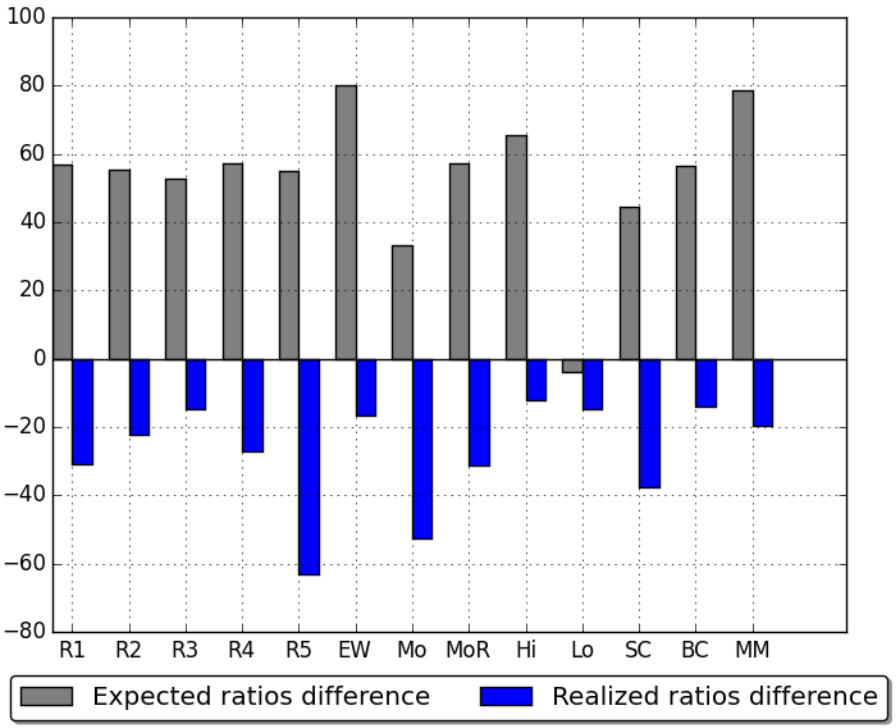


Figure 6.1: Average expected and realized ratios differences

The first column represents the mean evaluation function values for all the trader categories. As it is evident, the genetic strategy performs better than any other (10.73), apart from the low book/market one (14.62). They are, indeed, the only two strategies expecting a positive ratio to be observed during the following market cycle.

The second column, presenting the mean realized ratios, completely reverses the expectations: every trading strategy performs better than the predictions suggested. The only two traders who face a worse situation are the low book/market (-13.58) and the genetic (-28.22) ones.

The third and fifth columns of the table simply confirm what has been just said: the genetic strategy, *ex ante*, outperforms almost every other trading approach (positive expected ratio differences) while being consistently outperformed *ex post* (negative realized ratio differences).

The fourth and sixth column help understanding the change even better: strategies as the equally weighted one, that expected a ratio 747.34 % lower than the Ge one, actually realized a value 58.31 % higher.

The explanation to such behavior is to be found in table 6.2.

	excRetMean	excRiskMean	excRatioMean
R1	-9.05 e-04	-1.12 e-07	-39.34
R2	-2.07 e-03	-7.16 e-07	-48.01
R3	-7.50 e-04	7.59 e-09	-26.2
R4	-4.95 e-04	4.86 e-09	-37.42
R5	-1.42 e-03	-2.72 e-08	-79.54
EW	-4.22 e-04	-5.39 e-08	-53.79
Mo	-2.36 e-03	-1.23 e-05	-50.89
MoR	-7.50 e-04	-1.92 e-07	-46.29
Hi	-5.54 e-04	-2.25 e-07	-25.83
Lo	3.35 e-04	-2.29 e-09	21.36
SC	-1.40 e-03	1.03 e-07	-34.38
BC	-2.63 e-04	-6.07 e-08	-22.81
Ge	6.49 e-04	-1.75 e-07	29.11
MM	-4.36 e-04	-5.21 e-08	-54.98

Table 6.2: Excess returns, variances and ratios

Consider the second column: the difference between the expected and the realized variance is extremely small for any trader. Except from the momentum investor, any other individual performed an estimation error not greater than a value multiplied by 10^{-71}.

Errors in the portfolio's return estimation (first column), instead, are higher. This reflects on the difference between *ex ante* and *ex post* ratios: genetic and low book/market traders, the only ones who overestimated returns, turn out to be the only ones having overestimated ratios as well.

Even if, as explained above, the goal of the study is to observe the return-over-risk ratio dynamics, analysing portfolios expected and realized returns and variances gives some interesting insights.

	expRetDiff	retDiff	expRiskDiff	riskDiff
R1	7.41 e-04	-7.01 e-04	4.70 e-06	4.59 e-06
R2	7.95 e-04	-1.94 e-03	-1.36 e-05	-1.47 e-05
R3	6.94 e-04	-8.95 e-04	9.39 e-06	9.54 e-06
R4	7.61 e-04	-6.61 e-04	8.29 e-06	8.34 e-06
R5	7.70 e-04	-1.56 e-03	5.56 e-06	5.66 e-06
EW	6.97 e-04	-6.00 e-04	1.58 e-05	1.60 e-05
Mo	9.22 e-04	-2.37 e-03	-8.32 e-05	-7.29 e-05
MoR	8.75 e-04	-7.89 e-04	8.86 e-06	9.03 e-06
Hi	1.48 e-03	-2.488 e-04	-1.11 e-06	-1.05 e-06
Lo	-8.61 e-05	-4.733 e-04	7.47 e-06	7.62 e-06
SC	1.12 e-03	-1.33 e-03	-2.23 e-05	-2.24 e-05
BC	7.20 e-04	-4.92 e-04	1.13 e-05	1.14 e-05
Ge	0.0	0.0	0.0	0.0
MM	6.88 e-04	-6.26 e-04	1.58 e-05	1.59 e-05

Table 6.3: Expected and realized return and variance differences

Data from table 6.3, indeed, show the same tendency seen in the previous tables. *Ex ante*, the genetic strategy is expected to outperform all its competitors but the low book/market one, and this happens also for the expected return of the portfolio. *Ex post*, the genetic performance are poor in terms of evaluation function, and this reflects also on the realized returns, which on average are all greater than the genetic one.

6.2 Second run: small variance of prices, alternative GA settings

The second market run is almost identical to the first one. The only difference lays in the genetic algorithms set up: the real range mutator is substituted by a real Gaussian one, while the crossover is of the uniform type.

[1] In Python's outcomes, the notation e-07 stands for 10^{-7}, not e^{-7}.

The variation of stock prices is still limited by the range `(-1.0, 1.0)`.

The results, however, are different.

	expRatios	ratios	expRatiosDiff	%	ratiosDiff	%
R1	104.76	82.11	47.14	-31.03 %	46.92	-36.36 %
R2	108.65	104.45	43.25	-28.47 %	24.58	-19.05 %
R3	108.24	61.29	43.66	-28.75 %	67.74	-52.5 %
R4	98.65	101.78	53.25	-35.06 %	27.25	-21.12 %
R5	98.25	107.7	53.65	-35.32 %	21.33	-16.53 %
EW	150.48	146.9	1.42	-0.94 %	-17.87	13.85 %
Mo	93.54	82.5	58.36	-38.42 %	46.53	-36.06 %
MoR	44.59	72.86	107.31	-70.64 %	56.16	-43.53 %
Hi	43.94	62.83	107.97	-71.08 %	66.2	-51.3 %
Lo	92.07	91.54	59.83	-39.39 %	37.48	-29.05 %
SC	91.94	90.29	59.96	-39.47 %	38.74	-30.03 %
BC	48.83	25.16	103.07	-67.85 %	103.87	-80.5 %
Ge	151.9	129.03	0.0	0.0 %	0.0	0.0 %
MM	146.64	144.87	5.26	-3.46 %	-15.84	12.28 %

Table 6.4: Expected and realized ratios

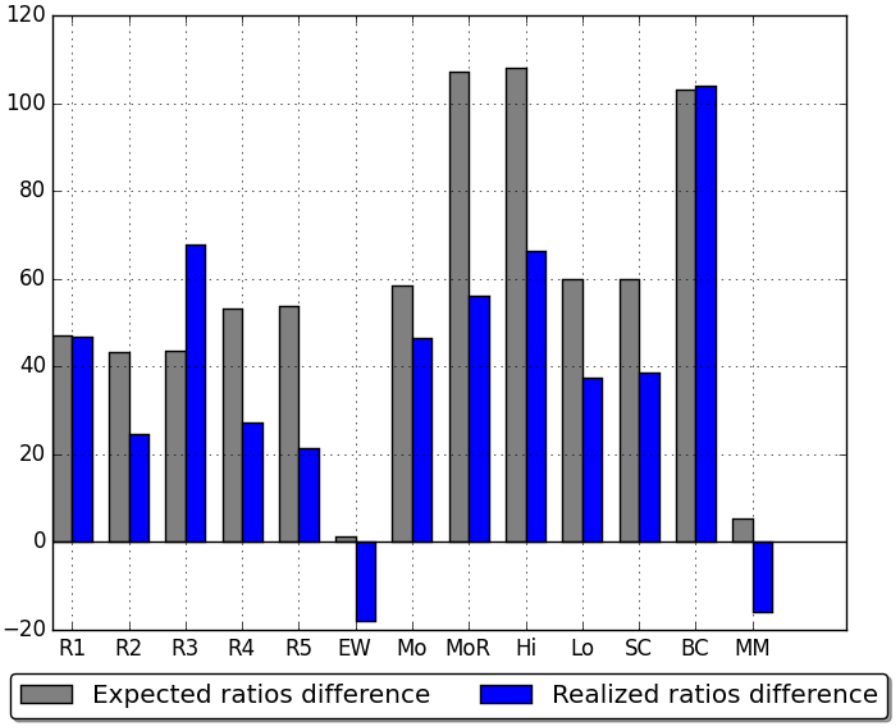


Figure 6.2: Average expected and realized ratios differences

From table 6.4 it is evident how the different behavior of prices in the simulation influenced the general performance of the traders. Differently from the first run, both expected and realized ratios are positive. Not differently from before, however, genetic algorithms outperform any other strategy in terms of expected values.

The main difference between the two simulations is located in the second column of the table: this time, indeed, the genetic approach predicted much more accurately the values, so that the *ex ante* (151.9) and the *ex post* (129.03) ratios present a distance equal to 16.28 %. The case-one difference, instead, equals -363.00 %.

Furthermore, this greater accuracy reflects also into better *ex post* performances: only the equally weighted trader and the market maker are able to beat the genetic strategy, while all the other traders are outperformed, featuring quite high ratios differences.

Not surprisingly, this is confirmed by table 6.5.

	excRetMean	excRiskMean	excRatioMean
R1	5.031 e-04	3.83 e-08	24.0
R2	-1.09 e-04	8.72 e-08	1.36
R3	2.45 e-04	2.76 e-07	45.63
R4	6.30 e-04	-8.72 e-08	3.09
R5	-4.29 e-04	-4.02 e-07	-12.92
EW	-1.53 e-05	5.47 e-08	-0.97
Mo	4.490 e-04	6.38 e-08	-0.22
MoR	4.24 e-03	4.74 e-06	-32.87
Hi	-4.68 e-04	1.50 e-07	-19.29
Lo	-9.45 e-05	3.19 e-07	-3.43
SC	-5.71 e-05	3.18 e-07	-2.26
BC	3.03 e-04	1.12 e-07	15.98
Ge	1.33 e-04	4.13 e-08	11.27
MM	-3.14 e-05	4.73 e-08	-2.35

Table 6.5: Excess returns, variances and ratios

While the ability to predict returns has not improved *per se*, a different ability to individuate proper portfolio compositions results into a lower error in ratio estimation which, this time, goes no more against the general tendency.

The analysis of return and variance differences, in this case, highlights how a good return-on-risk performance is not necessarily captured by the return values. Look at table 6.6.

	expRetDiff	retDiff	expRiskDiff	riskDiff
R1	1.46 e-04	3.51 e-04	-4.46 e-06	-4.50 e-06
R2	1.55 e-04	-1.91 e-04	-6.56 e-06	-6.45 e-06
R3	1.32 e-04	-4.47 e-06	-4.57 e-05	-1.52 e-05
R4	2.56 e-04	5.05 e-04	-1.13 e-05	-1.15 e-05
R5	2.05 e-04	-4.05 e-04	-7.66 e-06	-7.71 e-06
EW	1.81 e-04	-5.31 e-05	1.17 e-06	1.14 e-06
Mo	-4.00 e-04	-3.71 e-05	-2.06 e-05	-2.07 e-05
MoR	8.82 e-04	4.35 e-03	-1.03 e-03	-1.04 e-03
Hi	9.45 e-04	1.98 e-04	-1.00 e-05	-1.02 e-05
Lo	-8.73 e-03	-1.11 e-03	-1.73 e-05	-1.73 e-05
SC	-8.80 e-04	-1.08 e-03	-1.74 e-05	-1.73 e-05
BC	8.83 e-04	1.05 e-03	-8.18 e-06	-8.21 e-06
Ge	0.0	0.0	0.0	0.0
MM	2.12 e-04	-4.26 e-05	1.10 e-06	1.07 e-06

Table 6.6: Expected and realized return and variance differences

While, accordingly to the ratios trend, the equally weighted and the market maker strategies dominate the genetic one in terms of realized return, this happens also for six more strategies (R2, R3, R5, Mo, Lo, SC) that performed worse. This is explained by the two last columns of the table, that highlight how the genetic portfolio consistently scores a better variance value. The only two portfolios that obtained an average smaller risk, not surprisingly, are EW and MM.

6.3 Third run: high variance of prices

The third run goes back to the previous genetic algorithm configuration: together with the roulette wheel selection method, the real range mutator and the two-point crossover are exploited again.

What is highly different, here, is the price oscillation: the range of variation, for each trader, is set equal to `(-8.0, 10.0)`.

Results from table 6.7, now, show similarities with the first run.

	expRatios	ratios	expRatiosDiff	%	ratiosDiff	%
R1	10.61	8.57	7.26	-40.64 %	-0.16	1.92 %
R2	10.88	11.99	6.99	-39.13 %	-3.58	42.58 %
R3	10.66	11.2	7.21	-40.33 %	-2.79	33.18 %
R4	10.11	7.46	7.76	-43.41 %	0.95	-11.32 %
R5	9.52	7.71	8.35	-46.74 %	0.7	-8.28 %
EW	14.57	15.68	3.3	-18.48 %	-7.27	86.52 %
Mo	7.19	10.39	10.68	-59.77 %	-1.99	23.63 %
MoR	6.51	10.54	11.36	-63.58 %	-2.13	25.31 %
Hi	1.84	11.5	16.03	-89.69 %	-3.09	36.73 %
Lo	9.93	2.88	7.94	-44.44 %	5.53	-65.75 %
SC	3.61	6.77	14.26	-79.82 %	1.64	-19.48 %
BC	9.46	8.28	8.41	-47.08 %	0.12	-1.48 %
Ge	17.87	8.41	0.0	-0.0 %	0.0	-0.0 %
MM	14.54	15.3	3.33	-18.64 %	-6.9	82.03 %

Table 6.7: Expected and realized ratios

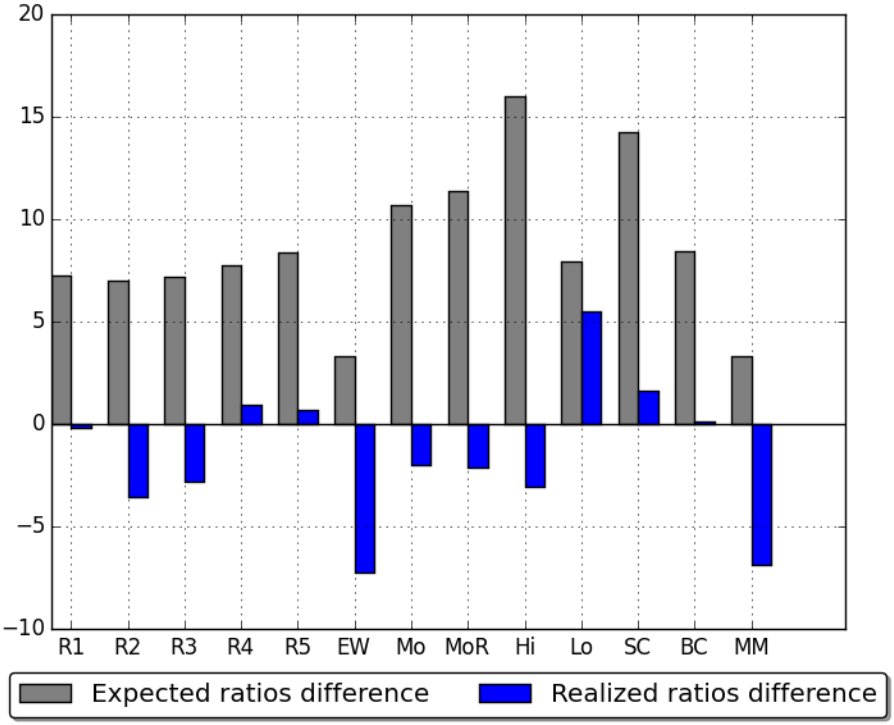


Figure 6.3: Average expected and realized ratios differences

Once again, when considering the expected values, the genetic trader outperforms all the other individuals. Its expected ratio, equal to 17.87, is higher than any other, resulting in no negative values in the third column.

Problems, however, raise again when observing the realized ratios: the variation now is high, equal to -52.94 %. The genetic strategy, *ex post*, re-

sults to be worse than 8 out of 13 strategies.

Again, the ability to predict (table 6.8) seems to be relevant.

	excRetMean	excRiskMean	excRatioMean
R1	4.88 e-04	1.55 e-04	0.23
R2	-5.40 e-03	9.78 e-06	-4.08
R3	-2.32 e-03	2.56 e-06	-3.04
R4	1.20 e-03	5.70 e-05	2.09
R5	3.47 e-03	-4.73 e-06	-0.93
EW	-1.12 e-03	2.04 e-06	-2.76
Mo	4.78 e-03	-1.02 e-04	-3.68
MoR	-6.75 e-02	-0.54 e-04	-4.16
Hi	-7.70 e-03	1.71 e-07	-9.09
Lo	4.56 e-03	1.11 e-05	5.14
SC	-4.69 e-03	-9.62 e-06	-3.12
BC	-9.83 e-04	6.28 e-06	-0.62
Ge	1.83 e-03	4.84 e-06	6.41
MM	-1.02 e-03	2.04 e-06	-2.43

Table 6.8: Excess returns, variances and ratios

The genetic trader overestimates returns with quite a big error. Random 4, random 5, low book/market and small cap traders, who performed worse (or only slightly better) in predicting the same value, have indeed been outperformed by the genetic strategy in terms of realized ratios.

The portfolios return and variance behavior, again, reflects the one observed for ratios.

	expRetDiff	retDiff	expRiskDiff	riskDiff
R1	1.70e-03	-3.86e-03	-3.26e-04	-3.43e-04
R2	8.93e-04	-6.11e-03	-3.76e-04	-3.84e-04
R3	9.03e-04	-3.11e-03	-2.34e-04	-2.34e-04
R4	1.29e-03	9.41e-04	-2.74e-04	-2.78e-04
R5	9.80e-04	2.84e-03	-4.14e-04	-4.24e-04
EW	1.09e-03	-2.41e-03	-2.33e-06	-2.58e-06
Mo	7.27e-04	5.13e-03	-1.20e-03	-1.11e-03
MoR	1.90e-02	-4.82e-02	-9.76e-02	-1.00e-01
Hi	4.71e-03	-6.35e-03	-4.82e-04	-4.81e-04
Lo	-2.49e-03	9.05e-04	-5.33e-04	-5.30e-04
SC	1.71e-03	-5.76e-03	-8.90e-04	-8.90e-04
BC	7.32e-04	-2.15e-03	-2.92e-04	-2.91e-04
Ge	0.0	0.0	0.0	0.0
MM	1.04e-03	-2.36e-03	-6.57e-06	-6.85e-06

Table 6.9: Expected and realized return and variance differences

The expected return values from table 6.9, apart from the low book/market case, are all dominated by the genetic one. The realized values, however,

invert the tendency, over-performing the algorithm in nine cases. Nonetheless, the genetic portfolio constantly outperformed its competitors in terms of risk, both *ex ante* and *ex post*.

6.4 Fourth run: high variance of prices, alternative GA settings

As already happened with the first two runs, the third and the fourth ones are identical in terms of prices dynamics: the range in which they can change is still `(-8.0, 10.0)`.

The modifications lie in the genetic algorithm's settings, that feature again a real Gaussian mutator, together with a uniform crossover technique.

Table 6.10 shows the consequent ratios values.

	expRatios	ratios	expRatiosDiff	%	ratiosDiff	%
R1	8.34	2.52	14.56	-63.59 %	10.66	-80.91 %
R2	6.57	1.87	16.33	-71.3 %	11.31	-85.81 %
R3	8.12	1.1	14.77	-64.52 %	12.08	-91.65 %
R4	9.73	4.79	13.17	-57.5 %	8.39	-63.67 %
R5	8.61	3.73	14.28	-62.38 %	9.45	-71.7 %
EW	11.02	-0.09	11.88	-51.89 %	13.27	-100.7 %
Mo	6.16	0.33	16.74	-73.09 %	12.85	-97.51 %
MoR	3.24	-0.89	19.66	-85.86 %	14.07	-106.76 %
Hi	1.3	-5.03	21.6	-94.32 %	18.21	-138.19 %
Lo	17.47	6.48	5.43	-23.7 %	6.69	-50.8 %
SC	1.11	-2.37	21.78	-95.14 %	15.55	-118.01 %
BC	17.52	2.25	5.38	-23.49 %	10.93	-82.94 %
Ge	22.9	13.18	0.0	-0.0 %	0.0	-0.0 %
MM	9.88	-0.39	13.01	-56.83 %	13.56	-102.93 %

Table 6.10: Expected and realized ratios

Not differently from the previous situations, the genetic strategy is able to ensure the highest expected mean ratio (22.9). What is interesting to notice, however, is that the results observed in section 6.2 are confirmed: when combining the real Gaussian mutator and the uniform crossover, *ex post* ratio values turn out to be better than when real range mutator and two-point crossover are exploited. While the general prediction performance is poor, genetic algorithms ensure a ratio which is more than two times bigger than the second best value (13.18, versus low book/market strategy's 6.48). This is not due to the high *ex ante* value alone, since traders who also expected to realize high ratio values performed poorly, showing drops of -62.91 % (low book/market) and -87.16 %.

The third and fourth column, containing not a single negative value, well summarize what just explained.

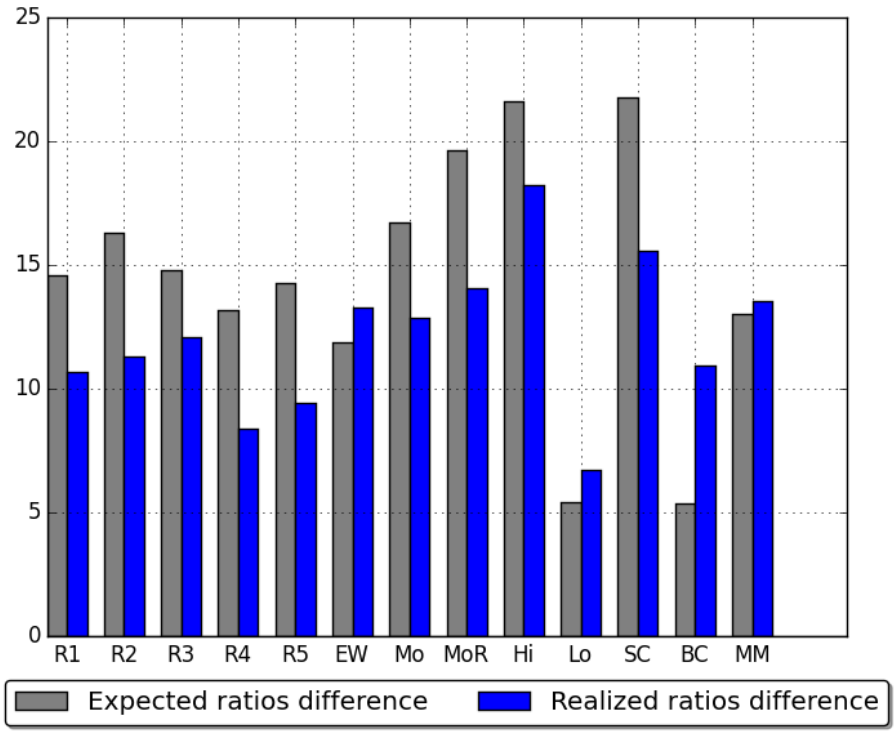


Figure 6.4: Average expected and realized ratios differences

Table 6.11 features other interesting aspects.

	excRetMean	excRiskMean	excRatioMean
R1	9.66 e-02	1.99 e-02	4.72
R2	4.3 e-03	4.01 e-06	3.69
R3	8.38 e-03	-1.75 e-05	6.86
R4	1.92 e-04	1.45 e-05	3.94
R5	4.41 e-03	2.01 e-06	3.72
EW	5.99 e-03	-1.71 e-06	10.26
Mo	1.09 e-02	-1.83 e-05	5.3
MoR	-2.63 e-04	-5.43 e-04	3.58
Hi	9.20 e-03	4.56 e-06	5.65
Lo	7.37 e-03	8.03 e-06	11.22
SC	5.38 e-03	-7.71 e-06	2.68
BC	8.50 e-03	2.26 e-06	16.06
Ge	3.82 e-03	7.54 e-07	10.8
MM	5.94 e-03	-1.93 e-06	9.36

Table 6.11: Excess returns, variances and ratios

In this case, indeed, the genetic composition leads to the smallest difference between predicted and realized portfolio's variance (7.54 e-07) and to the second smallest difference in portfolio's return (3.82 e-03).

Consider table 6.12 now.

	expRetDiff	retDiff	expRiskDiff	riskDiff
R1	2.65e-02	1.18e-01	-1.53e+00	-1.54e+00
R2	2.62e-03	4.90e-03	-1.08e-03	-1.10e-03
R3	2.57e-03	8.39e-03	-7.45e-04	-7.36e-04
R4	2.42e-03	1.04e-03	-8.46e-04	-8.56e-04
R5	2.30e-03	3.43e-03	-8.36e-04	-8.41e-04
EW	2.33e-03	5.53e-03	-2.01e-04	-2.02e-04
Mo	1.61e-03	9.98e-03	-9.41e-04	-9.48e-04
MoR	1.96e-04	-1.84e-03	-7.47e-02	-7.67e-02
Hi	6.46e-03	1.40e-02	-2.48e-03	-2.49e-03
Lo	-2.41e-03	1.16e-03	-2.47e-04	-2.46e-04
SC	6.18e-03	1.01e-02	-1.95e-03	-1.96e-03
BC	-6.06e-04	3.99e-03	-1.48e-04	-1.49e-04
Ge	0.0	0.0	0.0	0.0
MM	2.53e-03	5.76e-03	-2.49e-04	-2.50e-04

Table 6.12: Expected and realized return and variance differences

The general tendency of the genetic algorithm to outperform all the other strategies is confirmed, even if Lo and BC expect higher returns and MoR realizes a higher return. In terms of variances, instead, the genetic trader obtained the best performance in both cases.

6.5 Fifth run: bull market, small variance

For the fifth run, I chose to give a precise direction to the market movements. Each selling price, indeed, can here vary inside the range `(-1.0, 2.0)`. The variance of the prices is still limited, but the market will feature a constant bull trend.

To start off, the genetic algorithm is endowed with the two-point crossover and the real range mutator.

This particular set up leads to results never observed so far (table 6.13).

Genetic expected ratio (388.22), for the first time, is not the best one in the market: equally weighted investors (410.96) perform the best, but also the market maker (403.42) is able to beat the algorithm.

Talking about the realized values, it seems that the persistent bull market had good effects on the general predicting ability of the traders: even after observing the ratio realizations, the rank does not change. Equally weighted investors show the best ex post value (360.44), followed by the market maker (358.24). The genetic algorithm could not beat them (323.49), but consistently and highly outperformed all the other individuals.

	expRatios	ratios	expRatiosDiff	%	ratiosDiff	%
R1	263.03	238.2	125.19	-32.25 %	85.29	-26.36 %
R2	247.0	234.57	141.22	-36.38 %	88.92	-27.49 %
R3	260.26	260.46	127.96	-32.96 %	63.02	-19.48 %
R4	222.1	170.96	166.12	-42.79 %	152.53	-47.15 %
R5	256.61	212.82	131.61	-33.9 %	110.66	-34.21 %
EW	410.96	360.44	-22.74	5.86 %	-36.96	11.42 %
Mo	218.62	192.84	169.6	-43.69 %	130.65	-40.39 %
MoR	192.15	166.7	196.07	-50.51 %	156.78	-48.47 %
Hi	101.98	140.93	286.24	-73.73 %	182.56	-56.43 %
Lo	186.04	122.04	202.18	-52.08 %	201.44	-62.27 %
SC	126.28	157.31	261.94	-67.47 %	166.18	-51.37 %
BC	164.51	126.64	223.71	-57.63 %	196.84	-60.85 %
Ge	388.22	323.49	0.0	-0.0 %	0.0	-0.0 %
MM	403.42	358.24	-15.2	3.92 %	-34.76	10.74 %

Table 6.13: Expected and realized ratios

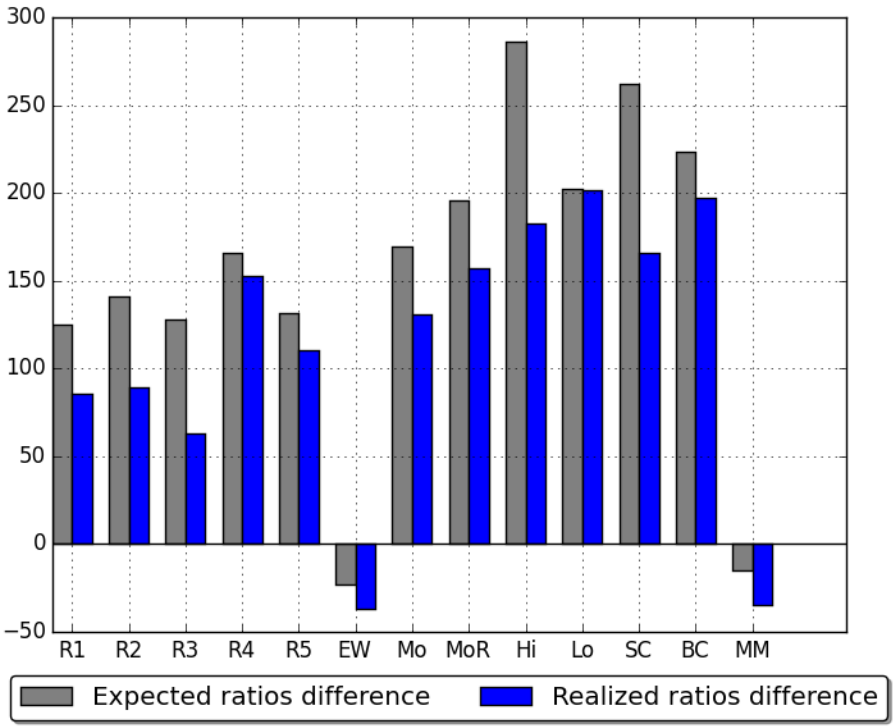


Figure 6.5: Average expected and realized ratios differences

Excess values from table 6.14, again, seem to have a correlation with the results:

	excRetMean	excRiskMean	excRatioMean
R1	-4.11 e-04	2.02 e-06	24.56
R2	6.91 e-05	1.24 e-06	8.54
R3	2.17 e-04	-1.04 e-06	3.49
R4	1.27 e-03	1.56 e-06	47.13
R5	4.13 e-04	-6.65 e-08	40.48
EW	3.63 e-04	2.17 e-08	43.53
Mo	-1.29 e-03	1.76 e-04	19.19
MoR	5.66 e-04	-1.31 e-08	28.31
Hi	-7.43 e-04	2.60 e-08	-36.56
Lo	1.24 e-03	7.92 e-08	54.33
SC	-3.79 e-04	-1.88 e-07	-21.54
BC	5.11 e-04	8.78 e-08	23.31
Ge	5.19 e-04	2.31 e-10	54.3
MM	3.30 e-04	2.05 e-08	38.66

Table 6.14: Excess returns, variances and ratios

The best performing strategies, indeed, all feature a good combination of excess return and excess variance: the equally weighted trader presents the fourth smaller excess return (3.63 e-04) and the fourth smaller excess variance (2.17 e-08), the market maker registered the third best excess return (3.30 e-04) and the third best excess variance (2.05 e-08), while the genetic strategy created a portfolio featuring the seventh smaller excess return (5.19 e-04) but, by far, the best excess variance (2.31 e-10). All the strategies with better excess returns, for example, obtained poor variance performances.

The return and variance behavior (table 6.15) is similar to the one observed in section 6.2:

	expRetDiff	retDiff	expRiskDiff	riskDiff
R1	2.10e-04	-8.76e-04	-4.60e-05	-4.64e-05
R2	1.35e-04	-7.86e-06	-8.53e-06	-8.68e-06
R3	8.27e-05	-2.91e-04	-5.59e-06	-5.39e-06
R4	7.15e-05	5.56e-04	-1.47e-05	-1.49e-05
R5	6.72e-05	-7.26e-05	-9.30e-06	-9.42e-06
EW	8.28e-05	-1.01e-04	7.01e-07	7.19e-07
Mo	-1.30e-04	7.32e-04	-6.63e-04	-6.77e-04
MoR	-1.22e-04	-1.96e-04	-1.01e-05	-1.00e-05
Hi	1.04e-03	-3.52e-04	-1.31e-05	-1.30e-05
Lo	-9.07e-04	-5.63e-05	-1.39e-05	-1.38e-05
SC	7.10e-04	-4.42e-04	-1.16e-05	-1.17e-05
BC	-2.29e-05	1.62e-04	-1.14e-05	-1.14e-05
Ge	0.0	0.0	0.0	0.0
MM	9.52e-05	-1.24e-04	5.88e-07	6.05e-07

Table 6.15: Expected and realized return and variance differences

The ranking observed with the ratios is not replicated here: Mo, MoR, Lo and BC outperform the algorithm *ex ante*, while EW and MM do not;

R1, R2, R3, R5, MoR, Hi and SC outperform the algorithm *ex post*. What determines the overall better performance of Ge is its portfolio's variance, which exactly replicates the ratio behavior: only EW and MM scored better.

6.6 Sixth run: bull market with small variance, alternative GA settings

The set up for the sixth market run is the same adopted in the previous section. However, the genetic trader now utilizes the real Gaussian mutator combined with the uniform crossover.

Table 6.16 present some new features:

	expRatios	ratios	expRatiosDiff	%	ratiosDiff	%
R1	238.34	190.48	122.94	-34.03 %	92.62	-32.72 %
R2	250.77	208.86	110.5	-30.59 %	74.24	-26.22 %
R3	238.19	184.33	123.08	-34.07 %	98.77	-34.89 %
R4	247.06	167.38	114.21	-31.61 %	115.72	-40.88 %
R5	249.58	219.67	111.7	-30.92 %	63.43	-22.41 %
EW	358.65	293.12	2.62	-0.73 %	-10.02	3.54 %
Mo	190.13	175.63	171.14	-47.37 %	107.47	-37.96 %
MoR	149.97	133.46	211.31	-58.49 %	149.64	-52.86 %
Hi	161.99	173.55	199.28	-55.16 %	109.54	-38.69 %
Lo	132.07	106.98	229.2	-63.44 %	176.12	-62.21 %
SC	132.07	106.98	229.2	-63.44 %	176.12	-62.21 %
BC	208.88	160.83	152.39	-42.18 %	122.27	-43.19 %
Ge	361.27	283.1	0.0	0.0 %	0.0	0.0 %
MM	351.71	286.76	9.56	-2.65 %	-3.66	1.29 %

Table 6.16: Expected and realized ratios

Differently from what has been observed so far, in a bull market different mutator and crossover have no relevant consequence. They are actually effective in the *ex ante* computations: in that case, the genetic ratio (361.27) is higher than the equally weighted (358.65) and market maker (351.71) ones. However, the difference is small: -0.73 % and -2.65 % respectively. Realized ratios, instead, penalize the genetic strategy again: exactly as in section 6.5, the equally weighted strategy turns out to be the best one (293.12), followed by the market maker (286.76). The genetic operator is anyway close to such results, performing just 3.54 % and 1.29 % worse respectively.

Again, excess values (table 6.17) follow this trend.

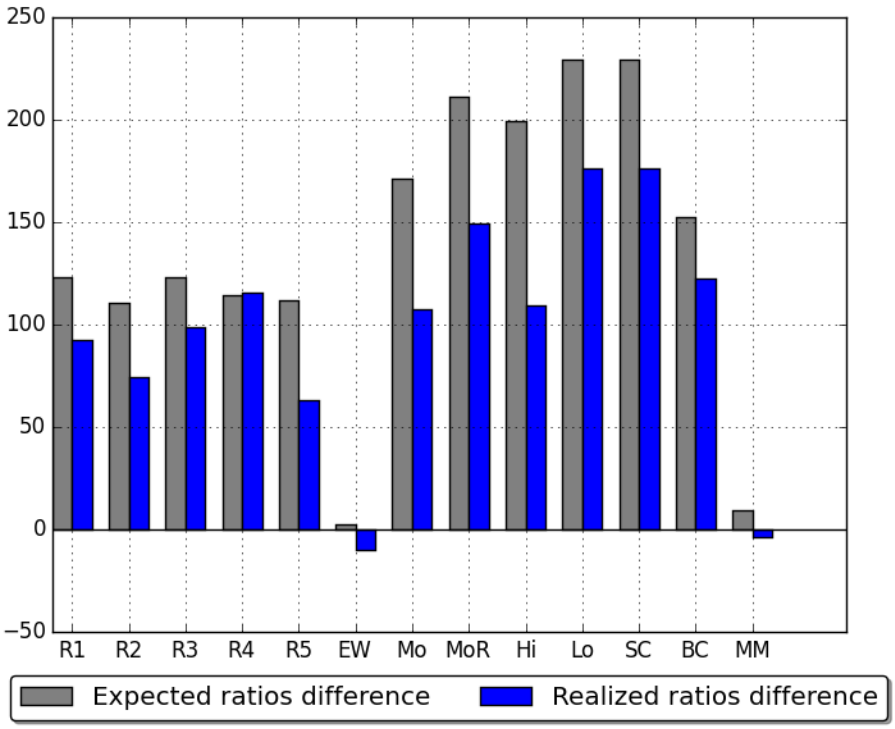


Figure 6.6: Average expected and realized ratios differences

	excRetMean	excRiskMean	excRatioMean
R1	3.99 e-05	-9.25 e-07	46.6
R2	-1.44 e-03	-1.19 e-06	41.92
R3	-4.13 e-03	1.50 e-05	56.86
R4	1.48 e-03	-1.81 e-07	81.86
R5	2.19 e-04	-2.19 e-07	34.58
EW	7.36 e-04	-4.28 e-08	66.82
Mo	1.02 e-03	-7.80 e-08	18.13
MoR	2.61 e-04	6.58 e-06	15.03
Hi	4.43 e-03	1.90 e-06	-15.1
Lo	1.03 e-03	-4.56 e-08	27.51
SC	1.03 e-03	-4.56 e-08	27.51
BC	8.25 e-04	-1.11 e-08	48.5
Ge	6.25 e-04	-8.47 e-08	69.84
MM	7.55 e-04	-4.00 e-08	66.8

Table 6.17: Excess returns, variances and ratios

The genetic portfolio still displays the most accurate variance (-8.47 e-08), while equally weighted traders and the market maker performed some o the best return predictions (7.36 e-04 and 7.55 e-04). In general, those three categories of traders, again, best combine return and variance excess values.

Table 6.18 well describes how returns and variances, alone, are not a good performance indicator when the object of study is the return-on-risk ratio:

	expRetDiff	retDiff	expRiskDiff	riskDiff
R1	-3.56e-04	-9.66e-04	-5.71e-05	-5.89e-05
R2	-2.80e-04	-2.41e-03	-8.82e-05	-9.12e-05
R3	2.09e-04	-4.40e-03	-1.81e-03	-1.84e-03
R4	-3.24e-04	4.54e-04	-9.14e-06	-9.27e-06
R5	-2.15e-04	-6.99e-04	-1.48e-05	-1.50e-05
EW	-2.18e-04	-2.09e-04	-5.06e-07	-4.97e-07
Mo	-4.82e-04	-1.80e-04	-2.70e-05	-2.72e-05
MoR	-8.15e-05	-1.05e-03	-4.50e-05	-4.52e-05
Hi	9.52e-04	4.53e-03	-1.06e-03	-1.08e-03
Lo	-1.38e-03	-1.16e-03	-2.82e-05	-2.82e-05
SC	-1.38e-03	-1.16e-03	-2.82e-05	-2.82e-05
BC	3.11e-04	4.00e-04	-5.85e-06	-5.83e-06
Ge	0.0	0.0	0.0	0.0
MM	-2.42e-04	-2.22e-04	-7.89e-07	-7.80e-07

Table 6.18: Expected and realized return and variance differences

Here, indeed, not only the return values do not reflect the behavior observed with ratios but, even, EW and MM, the only two strategies to outperform Ge, display worse returns and variances, both *ex ante* and *ex post*. However, it must be noticed how all the EW and MM values feature the smallest discrepancies with respect to Ge, especially for the variances (the are the only one with differences of the order e-07). This reflects the ratio behavior, where EW, MM and Ge values are extremely close.

6.7 Seventh run: bull market, high variance

The seventh run follows the concepts adopted in sections 6.5 and 6.6: the market shows a constant bull trend but, this time, the oscillations are much more wide. The range of variation is, indeed, `(-5.0, 8.0)`.

At first, I test a genetic algorithm adopting the real range mutator and the two-point crossover.

Table 6.19 contains the resulting ratio values:

	expRatios	ratios	expRatiosDiff	%	ratiosDiff	%
R1	24.91	20.77	21.23	-46.01 %	14.29	-40.76 %
R2	24.01	23.3	22.13	-47.96 %	11.75	-33.53 %
R3	25.04	25.58	21.1	-45.73 %	9.48	-27.04 %
R4	23.85	24.23	22.3	-48.32 %	10.82	-30.88 %
R5	25.77	23.2	20.37	-44.14 %	11.85	-33.81 %
EW	33.76	28.3	12.39	-26.85 %	6.76	-19.27 %
Mo	20.74	10.68	25.4	-55.05 %	24.37	-69.53 %
MoR	13.77	10.97	32.38	-70.16 %	24.09	-68.71 %
Hi	10.87	15.09	35.27	-76.44 %	19.97	-56.96 %
Lo	26.52	16.13	19.63	-42.54 %	18.92	-53.98 %
SC	10.05	8.97	36.09	-78.22 %	26.09	-74.42 %
BC	34.06	28.06	12.08	-26.19 %	6.99	-19.95 %
Ge	46.14	35.05	0.0	-0.0 %	0.0	-0.0 %
MM	33.49	27.9	12.65	-27.42 %	7.16	-20.42 %

Table 6.19: Expected and realized ratios

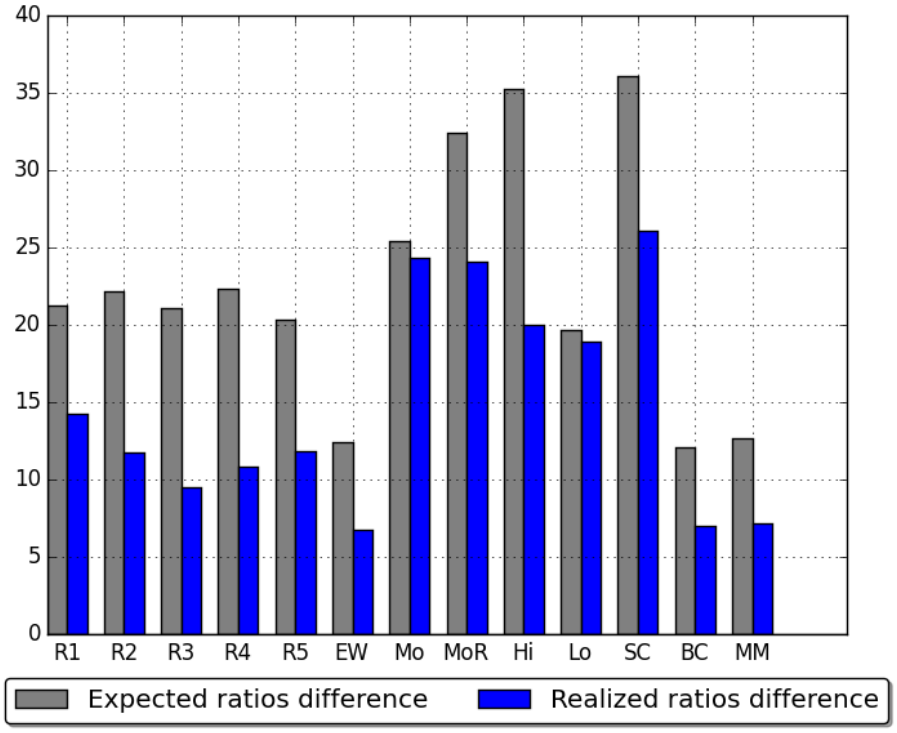


Figure 6.7: Average expected and realized ratios differences

A higher variance leads, differently from what observed in section 6.5, to lower ratios. Nonetheless, the performance ranking is only slightly different.

First of all, this time, the best expected ratio is associated with the genetic strategy (46.14). However, equally weighted (33.76) and market maker (33.49) still remain among the best traders. The *ex post* performances do

not change the situation, with the algorithm being able to outperform all its competitors, showing a difference of the 19.26% with the second best individual (35.05 versus the equally weighted trader's 28.3).

The excess values are collected in table 6.20:

	excRetMean	excRiskMean	excRatioMean
R1	-2.78 e-04	1.08 e-05	2.87
R2	2.22 e-03	5.10 e-06	-1.82
R3	6.88 e-04	5.52 e-06	-2.61
R4	2.91 e-03	-3.34 e-06	-1.89
R5	-4.82 e-03	-8.45 e-06	0.57
EW	7.67 e-04	3.12 e-06	3.68
Mo	5.26 e-03	1.22 e-05	8.59
MoR	1.60 e-02	4.05 e-04	2.11
Hi	-1.29 e-03	5.03 e-06	-5.2
Lo	2.89 e-03	1.11 e-05	8.56
SC	-8.68 e-05	8.72 e-06	0.32
BC	1.89 e-03	2.96 e-06	6.26
Ge	1.56 e-03	2.40 e-06	9.93
MM	7.84 e-04	3.13 e-06	3.82

Table 6.20: Excess returns, variances and ratios

First of all, it is evident how a higher range of variation for prices reflects into a higher discrepancy between predicted and realized returns and means.

Second, the genetic operator confirms its ability to predict its portfolio's variance, featuring the best excess value (2.04 e-06). The performance on the return, however is poor and reflects into a ratio fall of 9.93, the biggest observed.

Returns and variances (table 6.21), as usual, make the results not so evident:

	expRetDiff	retDiff	expRiskDiff	riskDiff
R1	1.09e-03	-5.93e-04	-1.34e-04	-1.33e-04
R2	1.19e-03	2.17e-03	-3.94e-04	-3.99e-04
R3	1.07e-03	4.80e-04	-1.68e-04	-1.70e-04
R4	9.98e-04	2.55e-03	-2.37e-04	-2.39e-04
R5	8.10e-04	-4.86e-03	-5.77e-04	-5.97e-04
EW	1.03e-03	3.80e-04	-2.87e-05	-2.84e-05
Mo	-3.14e-05	3.68e-03	-6.74e-04	-6.81e-04
MoR	8.98e-04	1.51e-02	-3.89e-02	-3.93e-02
Hi	2.83e-03	3.47e-04	-3.47e-04	-3.44e-04
Lo	-1.15e-03	6.11e-04	-6.06e-04	-6.10e-04
SC	2.38e-03	9.72e-04	-3.61e-04	-3.58e-04
BC	-6.26e-04	-4.96e-04	-7.39e-05	-7.35e-05
Ge	0.0	0.0	0.0	0.0
MM	1.02e-03	3.89e-04	-3.07e-05	-3.03e-05

Table 6.21: Expected and realized return and variance differences

Expected returns, indeed, are better for Mo, Lo and BC, while realized returns are higher for R1, R5 and BC. What justifies the observed ratios are the variance differences, all negative and quite high.

6.8 Eighth run: bull market with high variance, alternative GA settings

Again, the eighth run is dedicated to testing a different genetic approach in the same environment seen in section 6.7. Hence, prices are, on purpose, increasing and with a quite relevant oscillation, given by the range `(-5.0, 8.0)`. The genetic algorithm, instead, features a uniform crossover and a real Gaussian mutator.

Looking at the ratio values collected in table 6.22, results are aligned with what already seen.

The alternative algorithm parameters, once again, result in a good performance. The genetic portfolio displays the best expected ratio (46.6), dominating also the equally weighted strategy (36.95). As always observed, the realized values are lower, but this does not prevent the genetic portfolio to dominate all its rivals also *ex post* (35.58), with a difference of 12.49 % from the second best performing strategy.

	expRatios	ratios	expRatiosDiff	%	ratiosDiff	%
R1	29.53	22.27	17.07	-36.63 %	13.31	-37.4 %
R2	27.76	19.06	18.83	-40.42 %	16.52	-46.43 %
R3	28.39	25.14	18.21	-39.08 %	10.44	-29.33 %
R4	26.34	19.95	20.26	-43.47 %	15.63	-43.93 %
R5	26.98	25.77	19.61	-42.09 %	9.81	-27.57 %
EW	36.95	31.14	9.65	-20.71 %	4.44	-12.49 %
Mo	12.28	11.1	34.32	-73.65 %	24.48	-68.8 %
MoR	24.93	20.15	21.66	-46.49 %	15.43	-43.38 %
Hi	16.52	18.84	30.08	-64.55 %	16.74	-47.05 %
Lo	24.3	12.44	22.3	-47.86 %	23.14	-65.04 %
SC	11.16	11.88	35.44	-76.06 %	23.71	-66.63 %
BC	34.11	24.9	12.49	-26.8 %	10.68	-30.03 %
Ge	46.6	35.58	0.0	0.0 %	0.0	0.0 %
MM	35.3	29.59	11.3	-24.25 %	6.0	-16.85 %

Table 6.22: Expected and realized ratios

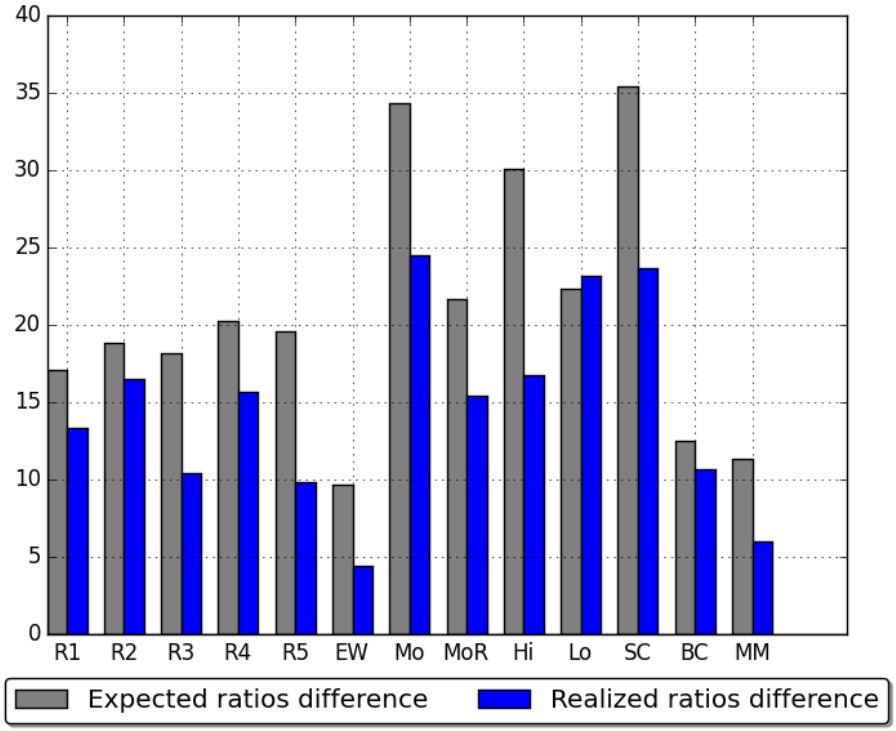


Figure 6.8: Average expected and realized ratios differences

In table 6.23, the genetic portfolio still shows one of the best abilities in predicting its variance, while problems persist in the accurate estimation of the portfolio's return.

	excRetMean	excRiskMean	excRatioMean
R1	2.51 e-03	-1.23 e-06	8.96
R2	5.00 e-03	1.18 e-05	9.39
R3	-1.02 e-04	8.92 e-06	2.5
R4	1.73 e-03	7.83 e-06	6.95
R5	1.11 e-03	-2.59 e-05	2.74
EW	1.74 e-03	3.62 e-06	6.19
Mo	5.81 e-02	3.34 e-04	0.06
MoR	2.34 e-03	4.98 e-06	5.14
Hi	7.57 e-04	1.83 e-05	-1.68
Lo	6.01 e-03	5.69 e-06	12.52
SC	-5.04 e-04	1.17 e-05	-0.57
BC	2.70 e-03	1.69 e-06	10.71
Ge	2.68 e-03	1.74 e-06	12.87
MM	1.77 e-03	3.79 e-06	6.0

Table 6.23: Excess returns, variances and ratios

The return and variance trends observed in table 6.24 are the one that most differ from the ratio ones:

	expRetDiff	retDiff	expRiskDiff	riskDiff
R1	-5.81e-04	-5.38e-04	-1.97e-04	-1.94e-04
R2	-6.08e-04	1.63e-03	-7.13e-04	-7.20e-04
R3	-3.89e-04	-2.86e-03	-3.40e-04	-3.43e-04
R4	-8.60e-04	-1.68e-03	-2.71e-04	-2.68e-04
R5	-1.69e-04	-1.58e-03	-2.90e-04	-2.91e-04
EW	-4.74e-04	-1.21e-03	-5.62e-05	-5.53e-05
Mo	-2.42e-03	5.12e-02	-5.82e-02	-5.91e-02
MoR	-4.22e-04	-6.01e-04	-2.22e-04	-2.21e-04
Hi	1.37e-03	-5.23e-04	-6.74e-04	-6.80e-04
Lo	-2.29e-03	1.00e-03	-2.60e-04	-2.58e-04
SC	-3.79e-04	-3.35e-03	-6.12e-04	-6.09e-04
BC	1.08e-04	1.15e-04	-5.85e-05	-5.84e-05
Ge	0.0	0.0	0.0	0.0
MM	-5.33e-04	-1.23e-03	-6.92e-05	-6.82e-05

Table 6.24: Expected and realized return and variance differences

In this case, indeed, the algorithm is able to outperform only two competitors *ex ante* (Hi and BC) and three of them *ex post* (R2, Mo and BC) in terms of return. Once again, the difference is made by the variance values, where the best performance is that of Ge itself.

6.9 Ninth run: constrained return

After having analyzed the behavior of genetic algorithms with a free evaluation function of the type `ret/risk`, the ninth and tenth market runs deal with more specific situations.

Here, the same evaluation function has to be maximized under a constraint: the portfolio's expected return can never be lower than 0.01. To achieve this, the program modification is simple. It is sufficient to modify the `if` condition written before the function in the following way:

```
if risk != 0 and ret >= 0.01:
```

Prices can widely vary, given the range `(-8.0, 10.0)`, while the genetic algorithm exploits a uniform crossover, Gaussian mutator and roulette wheel selection method.

Since the goal, now, is to obtain the lower risk possible given a minimum level of return, the analysis focuses on such values and, no more, on pure ratios.

Data are collected in table 6.25.

	expReturns	returns	expRisks	risks
R1	8.38e-03	7.65e-03	3.85e-03	7.57e-03
R2	1.88e-03	2.53e-01	-1.62e-02	2.50e-01
R3	8.79e-03	4.30e-03	-4.61e-03	4.27e-03
R4	7.92e-03	6.59e-04	7.04e-03	6.54e-04
R5	7.39e-03	7.04e-03	-6.38e-03	7.10e-03
EW	7.76e-03	4.55e-04	5.06e-03	4.52e-04
Mo	8.49e-03	1.41e-03	-1.08e-03	1.41e-03
MoR	7.83e-03	6.20e-03	7.46e-03	6.17e-03
Hi	5.09e-03	7.08e-04	4.85e-03	7.05e-04
Lo	1.06e-02	1.17e-03	5.39e-03	1.16e-03
SC	8.97e-03	1.58e-03	8.49e-03	1.57e-03
BC	6.54e-03	6.29e-04	7.34e-03	6.24e-04
Ge	9.87e-03	8.44e-04	8.20e-03	8.37e-04
MM	7.69e-03	4.62e-04	5.08e-03	4.59e-04

Table 6.25: Expected and realized returns and variances

First of all, looking at the first column, it can be seen how the constrain has been respected, obtaining an average expected return slightly lower than the one required (0.00987). Now, in order to test the goodness of the algorithm, it is possible to look at all the other agent's expected returns and, after that, analyze the associated expected risks. The general assumption is that a higher risk must be remunerated by a higher return hence, before scanning the data, this is exactly the expected situation. From this it follows that whenever a portfolio strategy displays an expected return which is lower than the genetic's one but, at the same time, features a higher variance, such strategy has been outperformed by the algorithm.

Table 6.25 reveals that such situation verifies more than once: R1, R2, R3, R5, Mo, MoR and SC all obtained expected return values lower than Ge, but all expected a higher risk from their portfolios. This is intuitively shown also in figure 6.9.

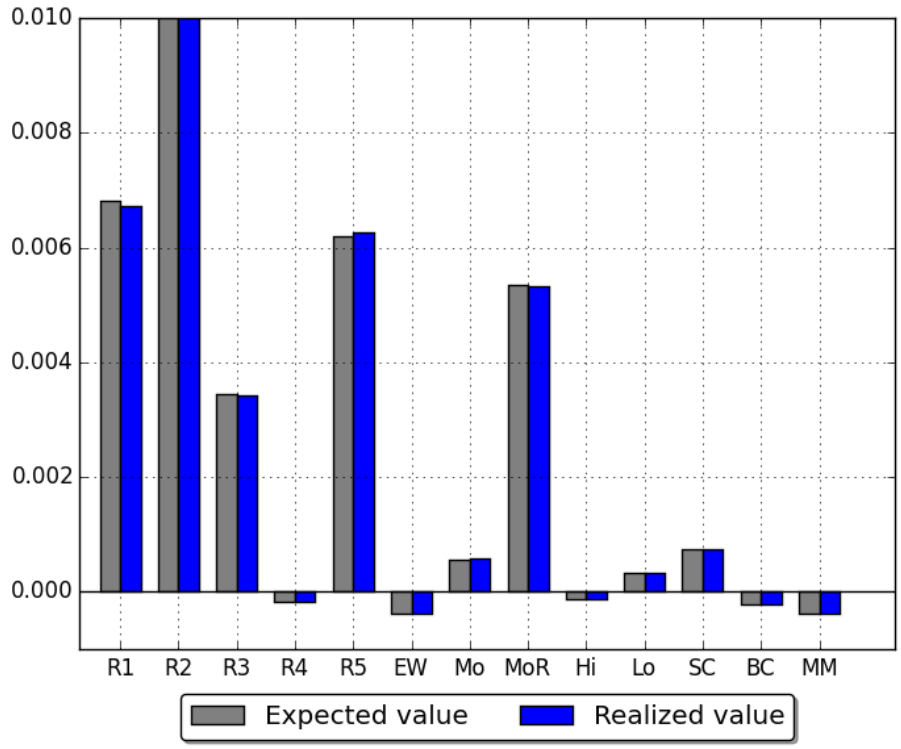


Figure 6.9: Average expected and realized variance differences

From there, it is also evident how the performances do not change after the actual portfolio values are observed: all the strategies that have been outperformed *ex ante*, have been dominate *ex post* as well.

Notice that, in figure 6.9, the bars referring to R2 are not completely shown because, given their great height, would make the scaling of the graph such that the data would be no more readable.

6.10 Tenth run: constrained variance

The tenth run maintains all the settings used in section 6.9, changing the optimization constraint: this time, the algorithm is ask to maintain the portfolio's variance below the target level of 6.00e-04. This translates into the following program string:

```
if risk != 0 and risk <= 0.0006:
```

Results are stored in table 6.26.

	expReturns	returns	expRisks	risks
R1	4.17e-03	4.81e-03	-5.87e-03	4.82e-03
R2	6.33e-03	1.28e-03	6.40e-03	1.27e-03
R3	5.99e-03	9.47e-04	4.85e-03	9.40e-04
R4	6.10e-03	2.41e-03	5.75e-03	2.39e-03
R5	3.04e-03	1.39e-01	-1.71e-02	1.38e-01
EW	6.19e-03	4.87e-04	3.91e-03	4.85e-04
Mo	4.55e-03	3.01e-02	1.03e-02	3.00e-02
MoR	4.62e-03	1.45e-03	5.51e-03	1.45e-03
Hi	2.43e-03	1.19e-03	4.05e-03	1.19e-03
Lo	1.00e-02	1.98e-03	3.04e-03	1.95e-03
SC	2.92e-03	1.22e-03	5.50e-03	1.22e-03
BC	9.28e-03	1.65e-03	-6.77e-04	1.63e-03
Ge	7.68e-03	4.26e-04	2.00e-03	4.25e-04
MM	6.36e-03	5.11e-04	3.94e-03	5.09e-04

Table 6.26: Expected and realized returns and variances

Given the new constraint, the analysis is now reversed: each time a portfolio scores a variance which is higher than the one obtained by Ge, it must also expect to obtain a higher return, or such different would not be justified. Whenever this is not true, the portfolio has been outperformed by Ge.

Similarly from section 6.9, this happens more than once: R1, R2, R4, R5, Mo, MoR, Hi, SC and MM all display higher risks which do not lead to higher returns.

Figure 6.10 shows it graphically.

This time, however, ex post results are extremely different: among all the above mentioned strategies, only R1 and R5 have been worse than Ge, while all the other traders obtained higher results (together with BC).

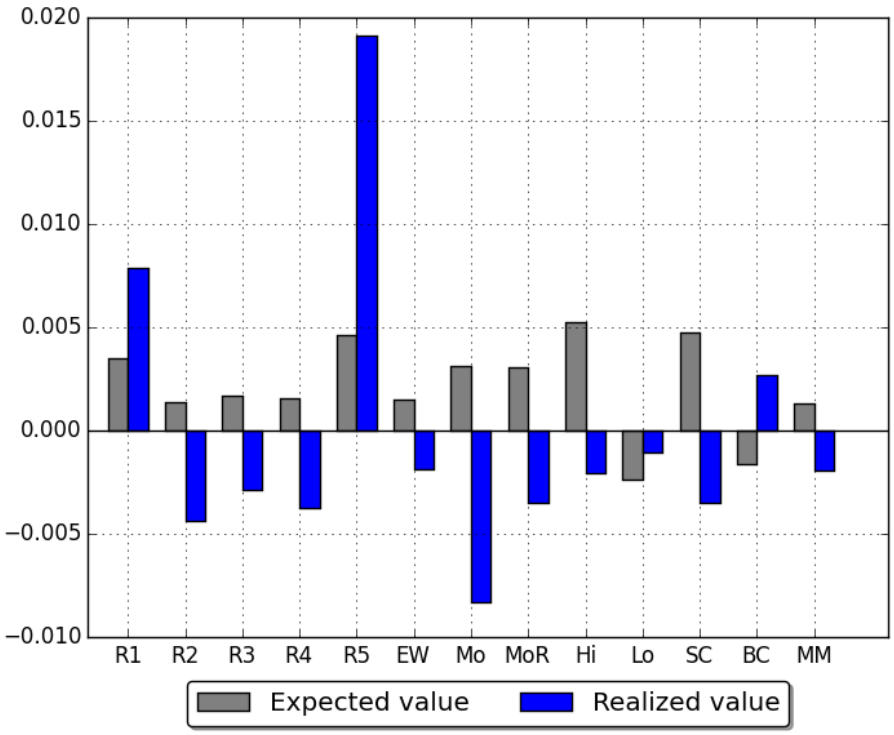


Figure 6.10: Average expected and realized return differences

Lastly, table 6.27 summarizes the results obtained from the first eight runs.

	expRatiosDiff	%	ratiosDiff	%
run 1	52.98	-493.76 %	-27.39	97.08 %
run 2	57.24	-37.68 %	38.69	-29.99 %
run 3	8.68	-48.59 %	-1.45	17.35 %
run 4	14.50	-63.35 %	12.07	-91.65 %
run 5	153.37	-39.51 %	112.54	-34.79 %
run 6	137.47	-38.05 %	97.86	-34.56 %
run 7	22.54	-48.84 %	14.81	-42.24 %
run 8	20.86	-44.77 %	14.64	-41.14 %

Table 6.27: Average expected and realized ratios differences

The first half of it contains the average expected ratios differences between the genetic portfolio and each other strategy: before any observation, the algorithms always outperformed the other approaches with good results, given that the smallest percentage difference equals 37.68 %. The second half of the table, instead, shows the *ex post* values, highlighting two aspects: the genetic realized ratios, in runs one and three, are not the best ones; moreover, excluding the fourth run, the measured percentage difference is

always lower than the *ex ante* value.

Chapter 7

Conclusion

The work is aimed at designing an application of genetic algorithms optimization to stock portfolios. In order to do so, two basic theoretical topics have to be understood: how genetic algorithms work and how portfolio theory develops an optimal assets selection.

The former task has been accomplished in chapter 1 by analyzing the heuristic building blocks, with particular attention devoted to the operations needed to obtain the best possible fitting solution. Crossover and mutations, together with appropriate selection methods, are crucial in order to develop the quality of the results. A suitable specification of the evaluation function, however, cannot be neglected: a perfectly designed algorithm trying to optimize the wrong object will inevitably lead to poor results.

The latter task has been faced in its basic form in chapter 2. Portfolio selection is a vast field containing a large amount of different strategies: Markowitz's mean-variance approach represents the best choice to understand any successive development. The reason for this is that it deals with the very basic elements that come into play during assets selection: the role of risk is well highlighted, and so is its relation with prices; dealing with utility functions, the theory adopts an intuitive approach which makes the logic behind it appear clear and, with a graphical representation, even intuitive. Hence, the mean-variance theory seemed enough to introduce the work, given that the program deals with the search for the optimal risky portfolio.

The practical approach to the main goal started with an accurate analysis of genetic algorithms in Python (chapter 3). The pyevolve library has been illustrated by means of a series of practical applications highlighting its main features. The commands appear to be intuitive and immediate, not requiring an excessive programming. The user, indeed, is left with the general core questions: choosing the proper set up turns out to be crucial for a good final performance, so elements such as selection methods, crossover techniques

and mutators should be wisely selected, depending on the task being faced.

The greater attention, however, needs to be devoted to the evaluation function specification. It is, indeed, the only element able to actually link the theoretical problem with the practical genetic optimization. It may be challenging to work out its appropriate specification but, once the obstacle is overcome, the evaluation function turns out to be a powerful tool, enabling the user to face a huge variety of issues.

The practical part of the work begins with a simple application of genetic algorithms to real market data (chapter 4) and continues with the design of the artificial stock market (chapter 5). Consistently with the possibilities offered by such simplified version of a real market, a series of trading strategies have been specified: genetic portfolios compete with random, equally weighted, momentum, momentum reversal, high book/market, low book/-market, small cap and big cap strategies. In addition, to a market maker is assigned the role of distributing the total amount of assets among investors and, in the following cycles, to maintain an acceptable level of liquidity. Nonetheless, it must be noticed, strategies may be sub-efficient due to an impossibility to operate all the desired transactions.

Operations are managed through the creation of usual buying and selling books, allowing to match the best demand and the best supply and to modify each trader's portfolio accordingly. Transactional data is stored and later used to compute a series of useful indicators such as stocks official prices as well as assets and portfolios returns and risks.

Before moving on, a remark should be made on values predictability: the genetic portfolio, indeed, is the only one to be constructed based on return and variance expected values. This means that the degree of accuracy of such predictions is able to influence future buying and selling choices, exposing the strategy to an additional element of risk. All its competitors, instead, work with present quantities (book-to-market ratio, capitalization, realized past returns) or without any need to observe any indicator at all.

Keeping this in mind, eight different set ups of the market have been designed and tested in chapter 6. The elements that have been manipulated are mainly two: the dynamics of price changes and the genetic algorithms settings. The evaluation function, instead, have been maintained the same: the simple `ret/risk` ratio has been investigated. The choice has been made because, despite its simplicity, such specification turns out to be a powerful one. Maximizing such a ratio is equivalent to search for the portfolio promising the highest return possible for a given level of risk (or viceversa).

What emerged from the experiments gave quite clear signals.

First of all, the *ex ante* effectiveness of the genetic strategy revealed:

in all eight cases the portfolio consistently scored the best return-over-risk ratio, with average differences never under 37.68 %. More in detail, only three strategies have been able to perform a better *ex ante* score, out of 104 possibilities.

A second result emerging clearly is that the *ex post* ratio values almost never scored better than the expectations. Expected indicators consistently dominated their observed counterparts, recalling the topic hinted above: predictability plays a central role in portfolio selection, and basing a strategy on inaccurate expected values can lead to extremely different *ex ante* and *ex post* results. Emblematic, in this sense, are the experiments run in sections 6.1 and 6.3: in both cases an extremely positive situation, featuring expected returns always higher than the competitors did, have been completely reversed by observations, that led to constant under-performances.

Third, it seems to emerge a relation between prices dynamics and goodness of the genetic strategy: the algorithms, indeed, performed better when markets featured an increasing trend, rather than a neutral one.

Analyzing the competing genetic algorithms, instead, it appears clear how real Gaussian mutators and uniform crossovers performed better than real range and two-point ones respectively. In sections 6.2 and 6.4, indeed, the alternative set up led to positive *ex post* performance, whereas sections 6.1 and 6.3 showed the genetic strategy to be dominated. Sections 6.5 and 6.6 also show evident differences: even without beating the equally weighted portfolio and the market maker *ex post*, the alternative algorithm improved the expected performances.

Lastly, sections 6.9 and 6.10 featured more specific tasks: the usual return-on-risk ratio had to be maximized while seeking a goal minimum (maximum) value for the return (risk). To prove the algorithm's effectiveness, the analysis of competing strategies obtaining similar, or even lower, levels of return, revealed a risk higher than the genetic one. The same reasoning applies to the constrained risk case, where similar portfolios scored lower returns.

Although the encouraging results, more experiments could (and should) be run, featuring different set ups, which may lead to antithetic conclusions. Unfortunately, this was not possible due to limited computational power.

Furthermore, an obvious developing for the present work could be that of giving the artificial stock market a higher level of complexity. Shares, for instance, could pay dividends: this would influence their prices and hence, ultimately, the portfolio choices. Another modification could be represented by the addition of (at least) one risk-free asset, which would enable the traders to search not only for the optimal risky stocks combination but, also, for the best overall portfolio. Extremely interesting would also be to implement a life cycle profile, thus to link the optimization to an ideal consumption pattern over cycles.

The final development of the work could be individuated in the implementation of a genetic strategy on real market data. This is, with no doubt, the most evident application of the program, which may serve various types of goals (again, determined by the appropriate evaluation function); optimal households consumption and savings profile, thus to integrate the public pensions system; optimal liquidity management and protection against inflation; profit maximization given a precise profile of risk, which is a common requirement in more than one occasion (think of insurance companies investments). Genetic algorithms' potential is great and just needs to be exploited.

Appendix A

Tutorial by examples: complete Python codes

A.1 Example 1: a matrix containing only zeros

```
from pyevolve import *

def eval_func(genome):
    score = 0.0
    for i in xrange (genome.getHeight()):
        for j in xrange (genome.getWidth()):
            if genome[i][j] == 1:
                score += 0.0025
    return score

genome = G2DBinaryString.G2DBinaryString(20, 20)
"""genome.setParams(rangemin = 0, rangemax = 10)"""
genome.evaluator.set(eval_func)
genome.crossover.set(Crossovers.G2DBinaryStringXSingleHPoint)
genome.mutator.set(Mutators.G2DBinaryStringMutatorSwap)

ga = GSimpleGA.GSimpleGA(genome)
"ga.selector.set(Selectors.GRouletteWheel)"
ga.setMinimax(Consts.minimaxType["minimize"])
ga.setGenerations(200)
ga.evolve(freq_stats = 10)

print ga.bestIndividual()
```

A.2 Example 2: summing up to 150

```
from pyevolve import *

def eval_func(genome):
    tot = 0.0
    score = 0.0
    for value in genome:
        if value <= 10.0:
            tot += value
        if tot == 150.0:
            score += 1.0
    return score

genome = G1DList.G1DList(20)
genome.evaluator.set(eval_func)
genome.setParams(rangemin = 0.0, rangemax = 10.0,
                 bestrawscore = 1.0, rounddecimal = 1)

ga = GSimpleGA.GSimpleGA(genome)
ga.setGenerations(1000)
ga.setMutationRate(0.05)
ga.terminationCriteria.set(GSimpleGA.RawScoreCriteria)
ga.evolve(freq_stats = 20)

print ga.bestIndividual()
print ("The sum of the list's elements is"), sum(ga.bestIndividual())
```

A.3 Example 3: equations solving

Equation $-x^x + 2x^2 + x - 5 = -11$:

```
from pyevolve import *

def eval_func(genome):
    score = 0.0
    x = genome[0]
    if (-(x**x)+(2*x**2)+x-5) == -11:
        score +=1
    return score

def run_main():
    genome = G1DList.G1DList(1)
    genome.evaluator.set(eval_func)
    genome.setParams(bestrawscore = 1.0, rounddecimal = 1)
    genome.crossover.clear()
    genome.mutator.set(Mutators.G1DListMutatorRealGaussian)

    ga = GSimpleGA.GSimpleGA(genome)
    ga.setGenerations(500)
    ga.setPopulationSize(1000)
    ga.selector.set(Selectors.GRouletteWheel)
    ga.terminationCriteria.set(GSimpleGA.RawScoreCriteria)
    ga.evolve(freq_stats = 20)

    print ga.bestIndividual()
    print ("The solution to the equation is"), sum(ga.bestIndividual())

if __name__=="__main__":
    run_main()
```

Equation $x^4 + xy + y^4 = 1$:

```
from pyevolve import *
from operator import itemgetter

def eval_func(genome):
    score = 0.0
    x = genome[0]
    y = genome[1]
    if x**4 + x*y + y*4 == 1:
        score +=1
    return score


def run_main():

    genome = G1DList.G1DList(2)
    genome.evaluator.set(eval_func)
    genome.setParams(bestrawscore = 1.0000, rounddecimal = 4)
    genome.mutator.set(Mutators.G1DListMutatorRealGaussian)

    ga = GSimpleGA.GSimpleGA(genome)
    ga.setGenerations(500)
    ga.setPopulationSize(2000)
    ga.selector.set(Selectors.GRouletteWheel)
    ga.terminationCriteria.set(GSimpleGA.RawScoreCriteria)
    ga.evolve(freq_stats = 20)

    print ga.bestIndividual()

if __name__=="__main__":
    run_main()
```

A.4 Example 4: functions maximization and minimization

Function $f(x, y) = 10x + 5y$, with $3 \leq x \leq 10$ and $4 \leq y \leq 7$

```
from pyevolve import *

def eval_func(genome):
    score = 0.0
    x = genome[0]
    y = genome[1]
    if x>=3 and x<=10 and y>=4 and y<=7:
        score = 10*x + 5*y
    return score


def run_main():

    genome = G1DList.G1DList(2)
    genome.evaluator.set(eval_func)
    "genome.setParams(bestrawscore = 135.0000, rounddecimal = 4)"
    genome.mutator.set(Mutators.G1DListMutatorRealGaussian)

    ga = GSimpleGA.GSimpleGA(genome)
    ga.setGenerations(500)
    ga.setPopulationSize(2000)
    ga.selector.set(Selectors.GRouletteWheel)
    "ga.terminationCriteria.set(GSimpleGA.RawScoreCriteria)"
    ga.evolve(freq_stats = 20)

    print ga.bestIndividual()

if __name__=="__main__":
    run_main()
```

Function $f(x) = x^2$

```
from pyevolve import *
from math import *

def eval_func(genome):
    x = genome[0]
    score = x*x
    return score

genome = G1DList.G1DList(1)
genome.evaluator.set(eval_func)
"genome.setParams(rangemin = -100, rangemax = 100)"
genome.initializator.set(Initializators.G1DListInitializatorReal)
genome.setParams(bestrawscore = 0.0000, rounddecimal = 4)
genome.mutator.set(Mutators.G1DListMutatorRealGaussian)
genome.crossover.clear()

ga = GSimpleGA.GSimpleGA(genome)
ga.setGenerations(500)
ga.setMinimax(Consts.minimaxType["minimize"])
ga.selector.set(Selectors.GRouletteWheel)
ga.setMutationRate(0.05)
ga.setPopulationSize(500)
ga.terminationCriteria.set(GSimpleGA.RawScoreCriteria)
ga.evolve(freq_stats=20)

print ga.bestIndividual()
```

Function $f(x, y) = 1 + x^2 + y^2$

```
from pyevolve import *
from math import *

def eval_func(genome):
    x = genome[0]
    y = genome[1]
    score = 1 + (x*x) + (y*y)
    return score

genome = G1DList.G1DList(2)
genome.evaluator.set(eval_func)
"genome.setParams(rangemin = -100, rangemax = 100)"
genome.initializator.set(Initializators.G1DListInitializatorReal)
genome.setParams(bestrawscore = 1.0000, rounddecimal = 4)
genome.mutator.set(Mutators.G1DListMutatorRealGaussian)

ga = GSimpleGA.GSimpleGA(genome)
ga.setGenerations(500)
ga.setMinimax(Consts.minimaxType["minimize"])
ga.selector.set(Selectors.GRouletteWheel)
"ga.setMutationRate(0.05)"
ga.setPopulationSize(500)
ga.terminationCriteria.set(GSimpleGA.RawScoreCriteria)
ga.evolve(freq_stats=20)

print ga.bestIndividual()
```

A.5 Example 5: a Black-Scholes application

```
from pyevolve import *
import numpy as np
from scipy.stats import norm

print("Hello")
print(" ")

K = input("Insert strike price value: ")
sigma = input("Insert sigma value: ")
T = input("Insert time to maturity value: ")
r = input("Insert risk-free interest rate value: ")
mi = input("Insert minimum underlying price value: ")
ma = input("Insert maximum underlying price value: ")
print(" ")

def eval_func(genome):
    S = genome[0]
    score = 0.0
    if S >= mi and S <= ma:
        d1 = (np.log (S / K) + ((r + (sigma**2 / 2)) * T)) /
                                (sigma * (np.sqrt(T)))
        d2 = d1 - (sigma * np.sqrt(T))
        score = S * norm.cdf(d1) - K * np.exp(-r * (T)) *
                                    norm.cdf(d2)
    return score


genome = G1DList.G1DList(1)
genome.crossover.clear()
genome.setParams(rangemin = 0, rangemax = ma)
genome.evaluator.set(eval_func)
genome.mutator.set(Mutators.G1DListMutatorRealGaussian)
genome.initializator.set(Initializators.G1DListInitializatorReal)

ga = GSimpleGA.GSimpleGA(genome)
ga.setGenerations(150)
ga.setPopulationSize(500)
"""ga.setMinimax(Consts.minimaxType["minimize"])"""
"ga.selector.set(Selectors.GRouletteWheel)"
ga.evolve(freq_stats = 20)

print ga.bestIndividual()
```

A.6 Example 6: creating a simple portfolio

```
from random import *
from operator import itemgetter
import matplotlib.pyplot as plt
from pyevolve import *

nBuyers = 1
buyersListP = []
buyersListB = []

nSellers = int(input("How many sellers? "))
sellersListP = []

# define price list for buyers #
for i in range(nBuyers):
    buyerP = round(uniform(105, 110), 3)
    buyersListP.append(buyerP)
    buyerB = round(uniform(400, 410))
    buyersListB.append(buyerB)

# define price list for sellers #
for i in range(nSellers):
    sellerP = round(uniform(90, 110), 3)
    sellersListP.append(sellerP)

print (" ")
print "Buyer price:", buyersListP
print (" ")
print "Sellers prices:", sellersListP
print (" ")

exePrices = []
buyP = itemgetter(0)(buyersListP)
buyB = itemgetter(0)(buyersListB)

##### genetic algorithm #####

def eval_func(genome):
    score = 0.0
    x = genome[0]
    y = genome[1]
    z = genome[2]
    w = genome[3]
```

```
    if 99.9 < ((x + y + z + w) / 4) < 100.1
    and (x + y + z + w) <= buyB:
        score +=1
    return score

setOfAlleles = GAllele.GAlleles()

for i in range(4):
  a = GAllele.GAlleleList(sellersListP)
  setOfAlleles.add(a)

genome = G1DList.G1DList(4)
genome.setParams(allele=setOfAlleles)
genome.evaluator.set(eval_func)
genome.mutator.set(Mutators.G1DListMutatorAllele)
genome.initializator.set(Initializators.G1DListInitializatorAllele)

ga = GSimpleGA.GSimpleGA(genome)
ga.setGenerations(500)
ga.setPopulationSize(200)
ga.evolve(freq_stats=50)

"print ga.bestIndividual()"

##### genetic algorithm #####

firstP = ga.bestIndividual()[0]
secondP = ga.bestIndividual()[1]
thirdP = ga.bestIndividual()[2]
fourthP = ga.bestIndividual()[3]

for i in range(nSellers):
    sellP = itemgetter(i)(sellersListP)

    if firstP < buyP and secondP < buyP
    and thirdP < buyP and fourthP < buyP:
        if sellP == firstP or sellP == secondP
        or sellP == thirdP or sellP == fourthP:
            if sellP <= buyP:
                exePrices.append(sellP)
                print "The buyer buys a stock from Seller",

if not exePrices:
    print (" ")
```

```
    print "One or more prices are too high for the buyer,
           could not complete he operation."
else:
    print (" ")
    print "The initial budget was:", buyB
    print "The total expenditure is:", sum(ga.bestIndividual())
    print "The execution prices are:", exePrices
    print "The average price paid is:", sum(ga.bestIndividual())/4
```

A.7 Example 7: creating a simple portfolio, case 2

```
from pyevolve import G1DList
from pyevolve import GSimpleGA
from pyevolve import Mutators
from pyevolve import Initializators
from pyevolve import GAllele
from random import *

nIRs = 1000
List = []

# create a list of interest rates #
for i in range(nIRs):
    ir = uniform(0.001, 0.2)
    List.append(ir)

# define the evaluation function #
def eval_func(genome):
    score = 0.0
    x = genome[0]
    y = genome[1]
    z = genome[2]
    w = genome[3]
    k = genome[4]
    a = genome[5]
    b = genome[6]
    if 0.04999 < ((x + y + z + w + k + a + b) / 7) < 0.05001:
        score +=1
    return score

def run_main():
   setOfAlleles = GAllele.GAlleles()

# define the possible values for the genomes #
   for i in range(7):
      a = GAllele.GAlleleList(List)
      setOfAlleles.add(a)

   genome = G1DList.G1DList(7)
   genome.setParams(allele=setOfAlleles)
   genome.evaluator.set(eval_func)
   genome.mutator.set(Mutators.G1DListMutatorAllele)
   genome.initializator.set(Initializators.G1DListInitializatorAllele)
```

```
    ga = GSimpleGA.GSimpleGA(genome)
    ga.setGenerations(500)
    ga.setPopulationSize(500)
    ga.evolve(freq_stats=50)

    print ga.bestIndividual()

# check the result #
    print "The average interest rate is", sum(ga.bestIndividual())/7

if __name__=="__main__":
     run_main()
```

Appendix B

ystockquote: complete Python codes

```
from pyevolve import *
import ystockquote
import numpy as np
np.seterr(invalid='ignore')

nInvestors = 1

# Get stocks data #
AAPL = ystockquote.get_historical_prices('AAPL', '2013-11-04', '2013-11-07')
BAC = ystockquote.get_historical_prices('BAC', '2013-11-04', '2013-11-07')
KMI = ystockquote.get_historical_prices('KMI', '2013-11-04', '2013-11-07')
PBR = ystockquote.get_historical_prices('PBR', '2013-11-04', '2013-11-07')
FB = ystockquote.get_historical_prices('FB', '2013-11-04', '2013-11-07')
GE = ystockquote.get_historical_prices('GE', '2013-11-04', '2013-11-07')
HAL = ystockquote.get_historical_prices('HAL', '2013-11-04', '2013-11-07')
INTC = ystockquote.get_historical_prices('INTC', '2013-11-04', '2013-11-07')
SDRL = ystockquote.get_historical_prices('SDRL', '2013-11-04', '2013-11-07')
SD = ystockquote.get_historical_prices('SD', '2013-11-04', '2013-11-07')

# Create historical prices lists #
pricesAAPL = []
a = 0
for key in AAPL:
    a = float(AAPL[key]['Open'])
    pricesAAPL.append(a)

pricesBAC = []
a = 0
```

```
for key in BAC:
    a = float(BAC[key]['Open'])
    pricesBAC.append(a)

pricesKMI = []
a = 0
for key in KMI:
    a = float(KMI[key]['Open'])
    pricesKMI.append(a)

pricesPBR = []
a = 0
for key in PBR:
    a = float(PBR[key]['Open'])
    pricesPBR.append(a)

pricesFB = []
a = 0
for key in FB:
    a = float(FB[key]['Open'])
    pricesFB.append(a)

pricesGE = []
a = 0
for key in GE:
    a = float(GE[key]['Open'])
    pricesGE.append(a)

pricesHAL = []
a = 0
for key in HAL:
    a = float(HAL[key]['Open'])
    pricesHAL.append(a)

pricesINTC = []
a = 0
for key in INTC:
    a = float(INTC[key]['Open'])
    pricesINTC.append(a)

pricesSDRL = []
a = 0
for key in SDRL:
    a = float(SDRL[key]['Open'])
```

```
    pricesSDRL.append(a)

pricesSD = []
a = 0
for key in SD:
    a = float(SD[key]['Open'])
    pricesSD.append(a)

# Compute log-returns, means and variances #
lnReturnsAAPL = []
lnReturnsBAC = []
lnReturnsKMI = []
lnReturnsPBR = []
lnReturnsFB = []
lnReturnsGE = []
lnReturnsHAL = []
lnReturnsINTC = []
lnReturnsSDRL = []
lnReturnsSD = []

meanAAPL = []
meanBAC = []
meanKMI = []
meanPBR = []
meanFB = []
meanGE = []
meanHAL = []
meanINTC = []
meanSDRL = []
meanSD = []

varianceAAPL = []
varianceBAC = []
varianceKMI = []
variancePBR = []
varianceFB = []
varianceGE = []
varianceHAL = []
varianceINTC = []
varianceSDRL = []
varianceSD = []

r = 0
T = len(pricesAAPL)
```

```
for t in range(T-1):
    presentMeans = []

    r = pricesAAPL[t+1]/pricesAAPL[t]
    lnr = np.log(r)
    lnReturnsAAPL.append(lnr)
    muAAPL = np.mean(lnReturnsAAPL)
    meanAAPL.append(muAAPL)
    presentMeans.append(muAAPL)
    varAAPL = np.var(lnReturnsAAPL)
    varianceAAPL.append(varAAPL)

    r = pricesBAC[t+1]/pricesBAC[t]
    lnr = np.log(r)
    lnReturnsBAC.append(lnr)
    muBAC = np.mean(lnReturnsBAC)
    meanBAC.append(muBAC)
    presentMeans.append(muBAC)
    varBAC = np.var(lnReturnsBAC)
    varianceBAC.append(varBAC)

    r = pricesKMI[t+1]/pricesKMI[t]
    lnr = np.log(r)
    lnReturnsKMI.append(lnr)
    muKMI = np.mean(lnReturnsKMI)
    meanKMI.append(muKMI)
    presentMeans.append(muKMI)
    varKMI = np.var(lnReturnsKMI)
    varianceKMI.append(varKMI)

    r = pricesPBR[t+1]/pricesPBR[t]
    lnr = np.log(r)
    lnReturnsPBR.append(lnr)
    muPBR = np.mean(lnReturnsPBR)
    meanPBR.append(muPBR)
    presentMeans.append(muPBR)
    varPBR = np.var(lnReturnsPBR)
    variancePBR.append(varPBR)

    r = pricesFB[t+1]/pricesFB[t]
    lnr = np.log(r)
    lnReturnsFB.append(lnr)
    muFB = np.mean(lnReturnsFB)
```

```
meanFB.append(muFB)
presentMeans.append(muFB)
varFB = np.var(lnReturnsFB)
varianceFB.append(varFB)

r = pricesGE[t+1]/pricesGE[t]
lnr = np.log(r)
lnReturnsGE.append(lnr)
muGE = np.mean(lnReturnsGE)
meanGE.append(muGE)
presentMeans.append(muGE)
varGE = np.var(lnReturnsGE)
varianceGE.append(varGE)

r = pricesHAL[t+1]/pricesHAL[t]
lnr = np.log(r)
lnReturnsHAL.append(lnr)
muHAL = np.mean(lnReturnsHAL)
meanHAL.append(muHAL)
presentMeans.append(muHAL)
varHAL = np.var(lnReturnsHAL)
varianceHAL.append(varHAL)

r = pricesINTC[t+1]/pricesINTC[t]
lnr = np.log(r)
lnReturnsINTC.append(lnr)
muINTC = np.mean(lnReturnsINTC)
meanINTC.append(muINTC)
presentMeans.append(muINTC)
varINTC = np.var(lnReturnsINTC)
varianceINTC.append(varINTC)

r = pricesSDRL[t+1]/pricesSDRL[t]
lnr = np.log(r)
lnReturnsSDRL.append(lnr)
muSDRL = np.mean(lnReturnsSDRL)
meanSDRL.append(muSDRL)
presentMeans.append(muSDRL)
varSDRL = np.var(lnReturnsSDRL)
varianceSDRL.append(varSDRL)

r = pricesSD[t+1]/pricesSD[t]
lnr = np.log(r)
lnReturnsSD.append(lnr)
```

```
muSD = np.mean(lnReturnsSD)
meanSD.append(muSD)
presentMeans.append(muSD)
varSD = np.var(lnReturnsSD)
varianceSD.append(varSD)

# Compute covariance matrices #
mat = np.matrix([lnReturnsAAPL, lnReturnsBAC, lnReturnsKMI,
                 lnReturnsPBR, lnReturnsFB, lnReturnsGE,
                 lnReturnsHAL, lnReturnsINTC, lnReturnsSDRL,
                 lnReturnsSD])
cov = np.cov(mat)

# Portfolio #

##### genetic algorithm #####
def eval_func(genome):
    score = 0.0
    ret = 0.0
    risk = 0.0

    for i in range(len(cov[:,1])):
        ret += genome[i] * presentMeans[i]

    for i in range(len(cov[:,1])):
        for i in range(len(cov[1,:])):
            risk += genome[i] * genome [j] * cov[i][j]

    score = ret/risk

    return score

genome = G1DList.G1DList(10)
genome.evaluator.set(eval_func)
genome.mutator.set(Mutators.G1DListMutatorRealGaussian)
genome.initializator.set(Initializators.G1DListInitializatorReal)

ga = GSimpleGA.GSimpleGA(genome)
ga.setGenerations(500)
pop = ga.getPopulation()
pop.scaleMethod.set(Scaling.SigmaTruncScaling)
ga.setPopulationSize(200)
ga.evolve(freq_stats=50)
##### genetic algorithm #####
```

```
best = ga.bestIndividual()
genWeights = []
for i in range(lenbest)):
    a = best[i]/purplesum(best)
    genWeights.append(a)

expGenReturn = 0
for i in range(len(genWeights)):
    expGenReturn += genWeights[i] * presentMeans[i]

genRisk = 0
for i in range(len(genWeights)):
    for i in range(len(genWeights)):
        genRisk += genWeights[i] * genWeights[j] * cov[i][j]

print (" ")
print "Genetic portfolio's weights are: ", genWeights
print "Genetic portfolio's expected return is: ", expGenReturn
print "Genetic portfolio's variance is: ", genRisk
print (" ")

# Random investors #
randWeights = np.random.dirichlet(np.ones(10), size=1)[0].tolist()

randReturn = 0
for i in range(len(randWeights)):
    randReturn += randWeights[i] * presentMeans[i]

randRisk = 0
for i in range(len(randWeights)):
    for i in range(len(randWeights)):
        randRisk += randWeights[i] * randWeights[j] * cov[i][j]

print "Random portfolio's weights are: ", randWeights
print "Random portfolio's expected return is: ", expRandReturn
print "Random portfolio's variance is: ", randRisk
print (" ")
```

Appendix C

Artificial stock market: complete Python code

Given the high number of parameters that can be manipulated, from the amount of investors to the type of evaluation function for the genetic algorithms, I present here just one version of the program, that can be easily modelled by simply modifying the desired variables.

```
import matplotlib.pyplot as plt
import numpy as np
from operator import itemgetter
import pickle
from pyevolve import *
from random import *
import time
np.seterr(divide='ignore', invalid='ignore')

timestr = time.strftime("%Y%m%d_%H%M%S")

nRandom = 20
nEqually = 20
nMomentum = 20
nReversal = 20
nHigh = 20
nLow = 20
nSmall = 20
nBig = 20
nGenetic = 1

nTotal = (nRandom + nEqually + nMomentum + nReversal +
          nHigh + nLow + nSmall + nBig + nGenetic + 1)
```

```
nShares = 5

budget = 1000
mmBudget = round(uniform(900000, 1100000), 3)

exePrices = []
for i in range(nShares):
    exePrices.append([])

exeQuant = []
for i in range(nShares):
    exeQuant.append([])

officialPrices = []
for i in range(nShares):
    officialPrices.append([])

lnReturns = []
for i in range(nShares):
    lnReturns.append([])

means = []
for i in range(nShares):
    means.append([])

variances = []
for i in range(nShares):
    variances.append([])

portfolios = []

oldWeights = []

initialPrices = []
for i in range(nShares):
    initialPrices.append(round(uniform(45, 145), 3))

initialQuant = []
for i in range(nShares):
    initialQuant.append(int(uniform(180000, 220000)))

    for t in range(10):
```

```
print "CYCLE ", t+1
print " "

traders = []
weights = []

if t == 0:
    # old weights
    for i in range(nTotal):
        w = []
        for j in range(nShares):
            w.append(0.0)
        oldWeights.append(w)

# portfolio
for i in range(nTotal - 1):
    ptf = []
    for j in range(nShares):
        ptf.append(0.0)
    portfolios.append(ptf)

# Random weights
for i in range(nRandom):
    we = []
    w = np.random.dirichlet(np.ones(nShares), size=1)[0].tolist()
    for z in range(len(w)):
        w[z] = int(w[z] * budget)
        we.append(w[z] - oldWeights[i][z])
    weights.append(we)

# Equal weights
for i in range(nEqually):
    we = []
    w = []
    for n in range(nShares):
        w.append(1./nShares)

    for n in range(len(w)):
        w[z] = int(w[z] * budget)
        we.append(w[z] - oldWeights[nRandom + i][z])
    weights.append(we)

# Momentum weights
for i in range(nMomentum):
```

```
    if t > 8 and t % 3 == 0:

        # rank stocks performances
        performances = []
        for j in range(nShares):
            a = []
            for s in range(t-9, t-1):
                a.append(lnReturns[j][s])
            m = np.mean(a)
            performances.append(m)
        p = sorted(performances, reverse = True)

        we = []
        w = []
        for n in range(len(p)):
            if performances[n] != 0 and performances[n] == p[0]
            or performances[n] != 0 and performances[n] == p[1]:
                w.append(1./2)
            else:
                w.append(0.)

        if sum(w) != 1.0:
            for n in range(len(w)):
                if w[n] != 0.0:
                    w[n] = 1.0

        for z in range(len(w)):
            w[z] = int(w[z] * trader[0])
            we.append(w[z] - oldWeights[nRandom + nEqually + i][z])
        weights.append(we)

    else:
        we = []
        for j in range(nShares):
            we.append(0.0)
        weights.append(we)

# Momentum reversal weights
for i in range(nReversal):
    if t > 8 and t % 3 == 0:

        # rank stocks performances
        p = sorted(performances)
```

```
            we = []
            w = []
            for n in range(len(p)):
                if performances[n] != 0 and performances[n] == p[0]
                or performances[n] != 0 and performances[n] == p[1]:
                    w.append(1./2)
                else:
                    w.append(0.)

            if sum(w) != 1.0:
                for n in range(len(w)):
                    if w[n] != 0.0:
                        w[n] = 1.0

            for z in range(len(w)):
                w[z] = int(w[z] * trader[0])
                we.append(w[z] - oldWeights[nRandom + nEqually +
                                            nMomentum + i][z])
            weights.append(we)

        else:
            we = []
            for j in range(nShares):
                we.append(0.0)
            weights.append(we)

    # High book/market weights
    for i in range(nHigh):
        if t > 0:

            # Rank b/m ratios
            BM = []
            for z in range(nShares):
                bm = initialPrices[z]/officialPrices[z][t-1]
                BM.append(bm)
            sortBM = sorted(BM, reverse = True)

            we = []
            w = []
            for n in range(len(BM)):
                if BM[n] == sortBM[0] or BM[n] == sortBM[1]:
                    w.append(1./2)
                else:
                    w.append(0.)
```

```
                for z in range(len(w)):
                    w[z] = int(w[z] * trader[0])
                    we.append(w[z] - oldWeights[nRandon + nEqually +
                              nMomentum + nReversal + i][z])
                weights.append(we)

            else:
                we = []
                for j in range(nShares):
                    we.append(0.0)
                weights.append(we)

        # Low book/market weights
        for i in range(nLow):
            if t > 0:

                # Rank b/m ratios
                sortBM = sorted(BM)

                we = []
                w = []
                for n in range(len(BM)):
                    if BM[n] == sortBM[0] or BM[n] == sortBM[1]:
                        w.append(1./2)
                    else:
                        w.append(0.)

                for z in range(len(w)):
                    w[z] = int(w[z] * trader[0])
                    we.append(w[z] - oldWeights[nRandon + nEqually +
                              nMomentum + nReversal + nHigh + i][z])
                weights.append(we)

            else:
                we = []
                for j in range(nShares):
                    we.append(0.0)
                weights.append(we)

    # Small cap weights
        for i in range(nSmall):

            # Rank capitalization
```

```
        Cap = []
        if t == 0:
            for j in range(nShares):
                a = initialQuant[j] * initialPrices[j]
                Cap.append(a)
        if t > 0:
            for j in range(nShares):
                a = initialQuant[j] * officialPrices[j][t-1]
                Cap.append(a)
        sortCap = sorted(Cap)

        we = []
        w = []
        for n in range(len(Cap)):
            if Cap[n] == sortCap[0] or Cap[n] == sortCap[1]:
                w.append(1./2)
            else:
                w.append(0.)

        for z in range(len(w)):
            w[z] = int(w[z] * budget)
            we.append(w[z] - oldWeights[nRandom + nEqually +
                      nMomentum + nReversal +
                      nHigh + nLow + i][z])
        weights.append(we)

    # Big cap weights
    for i in range(nBig):

        # Rank capitalization
        sortCap = sorted(Cap, reverse = True)

        we = []
        w = []
        for n in range(len(Cap)):
            if Cap[n] == sortCap[0] or Cap[n] == sortCap[1]:
                w.append(1./2)
            else:
                w.append(0.)

        for z in range(len(w)):
            w[z] = int(w[z] * budget)
            we.append(w[z] - oldWeights[nRandom + nEqually +
                                        nMomentum + nReversal +
```

```
                                        nHigh + nLow + nSmall + i][z])
        weights.append(we)

    # Genetic weights
    for i in range(nGenetic):
        if t > 4:

            ### GENETIC ALGORITHM ###
            def eval_func(genome):
                score = 0.0
                ret = 0.0
                risk = 0.0

                for j in range(nShares):
                    ret += genome[j] * means[j][t-1]

                for j in range(nShares):
                    for k in range(nShares):
                        risk += genome[j] * genome [k] * cov[j][k]

                if risk != 0:
                    score = ret / risk

                return score

            genome = G1DList.G1DList(5)
            genome.evaluator.set(eval_func)
            genome.mutator.set(Mutators.G1DListMutatorRealGaussian)
            genome.initializator.set(Initializators.G1DListInitializatorReal)

            ga = GSimpleGA.GSimpleGA(genome)
            ga.setGenerations(1000)
            pop = ga.getPopulation()
            pop.scaleMethod.set(Scaling.SigmaTruncScaling)
            ga.setPopulationSize(400)
            ga.evolve(freq_stats=250)
            ### GENETIC ALGORITHM ###

            best = ga.bestIndividual()
            we = []
            g = []
            for j in range(len(best)):
                a = best[j]/sum(best)
                g.append(a)
```

```
            for z in range(len(g)):
                g[z] = int(g[z] * budget)
                we.append(w[z] - oldWeights[nRandom + nEqually +
                                            nMomentum + nReversal +
                                            nHigh + nLow + nSmall +
                                            nBig + i][z])
            weights.append(we)

        else:
            we = []
            for j in range(nShares):
                we.append(0.0)
            weights.append(we)

    for i in range(len(weights)):
        trader = []
        trader.append(budget)

        for j in range(nShares):
            share = []
            if t == 0:
                share.append(initialPrices[j] + round(uniform(-5, 10), 3))
                share.append(initialPrices[j] + round(uniform(-5, 10), 3))
            if t > 0:
                share.append(officialPrices[j][t-1] +
                             round(uniform(-5, 10), 3))
                share.append(officialPrices[j][t-1] +
                             round(uniform(-5, 10), 3))
            if weights[i][j] > 0:
                share.append(weights[i][j])
            else:
                share.append(0)

            if weights[i][j] < 0:
                share.append(abs(weights[i][j]))
            else:
                share.append(0)
            if share[2] != 0:
                share.append(1)
            else:
                share.append(0)

            if share[3] != 0:
```

```
            share.append(1)
        else:
            share.append(0)
        trader.append(share)

    traders.append(trader)

# Market maker
marketMaker = []
marketMaker.append(mmBudget) #budget

for i in range(nShares):
    share = []
    if t == 0:
        share.append(0.0)
        share.append(initialPrices[i])
        share.append(0.0)
        share.append(initialQuant[i])
    if t > 0:
        share.append(officialPrices[i][t-1] + round(uniform(0, 10), 3))
        share.append(officialPrices[i][t-1] + round(uniform(0, 0), 3))
        share.append(initialQuant[i] -
                     portfolios[nRandom + nEqually + nMomentum +
                                nReversal + nHigh + nLow +
                                nSmall + nBig][i])
        share.append(portfolios[nRandom + nEqually + nMomentum +
                                nReversal + nHigh + nLow + nSmall +
                                nBig][i])
    if share[2] != 0:
        share.append(1)
    else:
        share.append(0)

    if share[3] != 0:
        share.append(1)
    else:
        share.append(0)
    marketMaker.append(share)

# portfolio
if t == 0:
    ptf = []
    for j in range(nShares):
        ptf.append(initialQuant[j])
```

```
    portfolios.append(ptf)

traders.append(marketMaker)

# Create books
buyLists = []
for i in range(1, nShares + 1):
    buy = []
    for j in range(len(traders)):
        a = [traders[j][i][0], traders[j][i][2]]
        if traders[j][i][2] != 0:
            a.append(1)
        else:
            a.append(0)
        buy.append(a)
    buyLists.append(buy)

sellLists = []
for i in range(1, nShares + 1):
    sell = []
    for j in range(len(traders)):
        a = [traders[j][i][1], traders[j][i][3]]
        if traders[j][i][3] != 0:
            a.append(1)
        else:
            a.append(0)
        sell.append(a)
    sellLists.append(sell)

buy = []
for i in range(len(buyLists)):
    b = sorted(buyLists[i], reverse = True)
    b = sorted(b, key = itemgetter(2), reverse = True)
    buy.append(b)

sell = []
for i in range(len(sellLists)):
    s = sorted(sellLists[i])
    s = sorted(s, key = itemgetter(2), reverse = True)
    sell.append(s)

# Match demand and supply
for i in range(nShares):
    exe = []
```

```
quant = []
while buy[i][0][0] >= sell[i][0][0] and buy[i][0][1] != 0
    and sell[i][0][1] != 0:
    exePrices[i].append(sell[i][0][0])
    exe.append(sell[i][0][0])
    exeQuant[i].append(min(buy[i][0][1], sell[i][0][1]))
    quant.append(min(buy[i][0][1], sell[i][0][1]))

    # modify portfolio
    for j in range(len(traders)):
        if buy[i][0][0] == traders[j][i+1][0]:
            portfolios[j][i] += (min(buy[i][0][1], sell[i][0][1]))
        if sell[i][0][0] == traders[j][i+1][1]:
            portfolios[j][i] -= (min(buy[i][0][1], sell[i][0][1]))

    buy[i][0][1] -=min(buy[i][0][1], sell[i][0][1])
    if buy[i][0][1] <= 0:
        buy[i][0][2] = 0
    sell[i][0][1] -= min((buy[i][0][1]), sell[i][0][1])
    if sell[i][0][1] >= 0:
        sell[i][0][2] = 0

    # Re-sort books
    b = sorted(buy[i], reverse = True)
    b = sorted(b, key = itemgetter(2), reverse = True)
    buy[i] = b
    s = sorted(sell[i])
    s = sorted(s, key = itemgetter(2), reverse = True)
    sell[i] = s

# Official price
totalWeights = 0
totalProd = 0
prod = 0
if quant:
    for j in range(len(exe)):
        totalWeights += quant[j]
        prod = exe[j] * quant[j]
        totalProd += prod

    offPrice = totalProd / totalWeights
    officialPrices[i].append(offPrice)

else:
```

```
            if officialPrices[i]:
                offPrice = officialPrices[i][t-1]
                officialPrices[i].append(offPrice)

            if not officialPrices[i]:
                offPrice = initialPrices[i]
                officialPrices[i].append(offPrice)

        # Log-return
        if t > 0:
            r = officialPrices[i][t] / officialPrices[i][t-1]
            lnr = np.log(r)
            lnReturns[i].append(lnr)

        else:
            lnr = 0
            lnReturns[i].append(lnr)

        # Mean
        mu = np.mean(lnReturns[i])
        means[i].append(mu)

        # Variance
        var = np.var(lnReturns[i])
        variances[i].append(var)

# Covariance matrix
mat = np.matrix(lnReturns)
cov = np.cov(mat)

# Portfolios percentages
percentages = []
for j in range(len(portfolios)):
    if sum(portfolios[j]) != 0:
        a = sum(portfolios[j])
        b = []
        for k in range(len(portfolios[j])):
            b.append(portfolios[j][k]/a)
            percentages.append(b)
    else:
        a = sum(portfolios[j])
        b = []
        for k in range(len(portfolios[j])):
            b.append(0.0)
```

```
            percentages.append(b)

    # Portfolios realized returns
    if t > 0:
        ptfReturns = []
        for j in range(len(oldPercentages)):
            ret = 0
            for k in range(nShares):
                ret += oldPercentages[j][k] * lnReturns[k][t]
            ptfReturns.append(ret)

    # Portfolios expected returns
    ptfExpReturns = []
    for j in range(len(percentages)):
        ret = 0
        for k in range(nShares):
            ret += percentages[j][k] * means[k][t]
        ptfExpReturns.append(ret)

    # Portfolios risks
    ptfRisks = []
    for j in range(len(percentages)):
        risk = 0
        for k in range(nShares):
            for h in range(nShares):
                risk += percentages[j][k] * percentages[j][h] * cov[k][h]
        ptfRisks.append(risk)

    oldWeights = portfolios
    oldPercentages = percentages
```

Bibliography

Baker (1985). *Adaptive selection methods for genetic algorithms*. In «Proceedings of the First International Conference on Genetic Algorithms and Their Applications», pp. 101-111.

Baker, J. E. (1987). *Reducing bias and inefficiency in the selection algorithm*. In «Proceedings of the Second International Conference on Genetic Algorithms and their Application», pp. 14-21.

Barricelli, N. A. (1954). *Esempi numerici di processi di evoluzione*. In «Methoods», pp. 45-68.

Bledsoe, W. W. (1961). *The use of biological concepts in the analytical study of systems*. Paper presented at ORSA-TIMS National Meeting, San Francisco.

Bodie, Z., Kane, A. and Marcus, A. (2013). *Investments*. McGraw-Hill, 10th ed.

Box, G. E. P. (1957). *Evolutionary operation: A method for increasing industrial productivity*. In «Journal of the Royal Statistical Society. Series C (Applied Statistics) 6», (2), pp. 81-101.

Bremermann, H. J. (1962). *Self-Organizing Systems*, chap. Optimization through evolution and recombination. Spartan Books, pp. 93-106.

Crosby, J. L. (1973). *Computer Simulation in Genetics*. John Wiley & Sons.

De Jong, K. A. (1975). *An Analysis of the Behavior of a Class of Genetic Adaptive Systems*. PhD Thesis, University of Michigan, Ann Arbor.

Eiben, A. E., Raué, P.-E. and Ruttaky, Z. (1994). *Genetic algorithms with multi-parent recombination*. In «Proceedings of the International Conference on Evolutionary Computation. The Third Conference on Parallel Problem Solving from Nature», pp. 78-87.

Fabozzi, F. J., Focard, S. M. and Kolm, P. N. (2006). *Financial Modeling of the Equity Market: From Capm to Cointegration*. John Wiley & Sons, 1st ed.

Fama, E. F. (1965). *Portfolio Analysis In a Stable Paretian Market.* In «Management Science», vol. 11(3), pp. 404-419.

— (1970). *Efficient Capital Markets: A Review of Theory and Empirical Work.* In «Journal of Finance», pp. 383-417.

Fogel, L. J., Owens, A. J. and Walsh, M. J. (1966). *Artificial Intelligence through Simulated Evolution.* John Wiley & Sons.

Forrest, S. (1985). *Scaling fitnesses in the genetic algorithm.* Documentation for Prisoners Dilemma and Norms Programs that Use the Genetic Algorithm. Unpublished manuscript.

Fraser, A. (1957). *Simulation of genetic systems by automatic digital computers.* In «Australian Journal of Biological Sciences», (10), pp. 484-491.

Fraser, A. S. and Burnell, D. (1970). *Computer Models in Genetics.* McGraw-Hill.

Friedman, G. J. (1959). *Digital simulation of an evolutionary process.* In «General Systems: Yearbook of the Society for General System Research», vol. 4, pp. 171-184.

Goldberg, D. E. (1987). *Simple genetic algorithms and the minimal deceptive problem.* In «Genetic Algorithms and Simulated Annealing». Morgan Kaufmann.

Holland, J. (1975). *Adaptation in natural and artificial systems.* University of Michigan Press.

— (1992). *Genetic Algorithms: Computer programs that "evolve" in ways that resemble natural selection can solve complex problems even their creators do not fully understand.* In «Scientific American», pp. 66-72.

Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* A Bradford Book.

Lintner, J. (1965). *The Valuation of Risk Assets and the Selection of Risky Investments in Stock Portfolios and Capital Budgets.* In «Review of Economics and Statistics», pp. 13-37.

Markowitz, H. (1952). *Portfolio Selection.* In «The Journal of Finance», vol. 7(1), pp. 77-91.
URL `https://www.math.ust.hk/~maykwok/courses/ma362/07F/markowitz_JF.pdf`

Mitchell, M. (1999). *An Introduction to Genetic Algorithms.* A Bradford Book, fifth ed.

Rechenberg, I. (1965). *Cybernetic solution path of an experimental problem*. Royal Aircraft Establishment, Library Translation No. 1122.

— (1973). *Evolutionsstrategie: Optimierung Technischer Systeme nach Prinzipien der Biologischen Evolution*. Frommann-Holzboog.

Schwefel, H.-P. (1975). *Evolutionsstrategie und numerische Optimierung*. PhD Thesis, Technische Universität Berlin.

— (1977). *Numerische Optimierung von Computer-Modellen mittels der Evolutionsstrategie*. Birkhäuser.

Sharpe, W. F. (1964). *Capital Asset Prices: A Theory of Market Equilibrium Under Conditions of Risk*. In «Journal of Finance», pp. 425-442.

Sivanandam, S. N. and Deepa, S. N. (2008). *Introduction to Genetic Algorithms*. Springer.

Tanese, R. (1989). *Distributed Genetic Algorithms for Function Optimization*. PhD Thesis, University of Michigan, Electrical Engineering and Computer Science Department.

Ting, C.-K. (2005). *On the Mean Convergence Time of Multi-parent Genetic Algorithms Without Selection*. In «Proceedings of the 8th European conference on Advances in Artificial Life», pp. 403-412.

Tobin, J. (1958). *Liquidity Preference as a Behavior Towards Risk*. In «Review of Economic Studies», pp. 65-86.

Turing, A. M. (1950). *Computing machinery and intelligence*. In «Mind», (236), pp. 433-460.
URL `http://mind.oxfordjournals.org/content/LIX/236/433`