TECHNISCHE
UNIVERSITÄT
WIEN

VIENNA
UNIVERSITY OF
TECHNOLOGY

DIPLOMARBEIT

Option Pricing by means of Genetic Programming

ausgeführt am Institut für
Computer Graphik und Algorithmen
der Technischen Universität Wien

unter Anleitung von a.o. Univ.-Prof. Dipl.-Ing. Dr.techn. Günther Raidl

und Univ.-Prof. Dr. Michael Hanke

durch

Andreas Heigl
Markt 115
5611 GROSSARL

Datum

Unterschrift

Abstract

This master thesis describes how to price options by means of Genetic Programming. The underlying model is the Generalized Autoregressive Conditional Heteroskedastic (GARCH) asset return process. The goal of this master thesis is to find a closed-form solution for the price of European call options where the underlying securities follow a GARCH process. The data are simulated over a wide range to cover a lot of existing options in one single equation.

Genetic Programming is used to generate the pricing function from the data. Genetic Programming is a method of producing programs just by defining a problem-dependent fitness function. The resulting equation is found via a heuristic algorithm inspired by natural evolution. Three different methods of bloat control are used. Additionally Automatic Defined Functions (ADFs) and a hybrid approach are tested, too. To ensure that a good configuration setting is used, preliminary testing of many different settings has been done, suggesting that simpler configurations are more successful in this environment.

The resulting equation can be used to calculate the price of an option in the given range with minimal errors. This equation is well behaved and can be used in standard spread sheet programs. It offers a wider range of utilization or a higher accuracy, respectively than other existing approaches.

Zusammenfassung

Diese Diplomarbeit beschreibt, wie Optionen mit Hilfe Genetischer Programmierung bewertet werden können. Das zugrunde liegende Modell nennt sich GARCH (Generalized Autoregressive Conditional Heteroskedastic) Renditeprozess. Das Ziel dieser Diplomarbeit ist eine geschlossene Formel, die als Ergebnis den Preis einer europäischen Kaufoption liefert, dessen dahinter liegende Wertpapier einem GARCH Prozess folgt. Die Daten werden innerhalb eines breiten Wertebereiches simuliert, um die meisten existierenden Optionen mit einer Formel bewerten zu können.

Die Formel wird mittels Genetischer Programmierung aus den Daten generiert. Genetische Programmierung ist eine Methode, bei der nur durch Definition einer zum Problem passenden Bewertungsfunktion vollständige Programme produziert werden können. Die Ergebnisgleichung wird schließlich mittels eines der Evolution ähnlichen Algorithmus gefunden. Drei verschiedene Methoden zum Bloat Control wurden verwendet. Zusätzlich wurden auch Automatisch Definierte Funktionen sowie ein hybrider Ansatz untersucht. Um sicherzustellen, dass eine gute Konfiguration gewählt wird, gibt es Vortests vieler verschiedener Konfigurationen. Es zeigt sich, dass in diesem Umfeld einfachere Konfigurationen erfolgreicher sind.

Die Ergebnisgleichung kann schließlich zur Errechnung der Optionspreise mit minimalem Fehler verwendet werden. Diese Gleichung verhält sich gut und kann auch in Standardtabellenkalkulationen verwendet werden. Im Vergleich mit anderen existierenden Ansätzen, bietet diese Gleichung eine weitere Verwendbarkeit beziehungsweise eine höhere Genauigkeit.

Acknowledgements

I have to thank my family, my professors and all my friends. Special thanks to Dr. Hanke, who has helped me to find this interesting topic of research and to Dr. Raidl, who has showed me how to write a good master thesis. All the brave programmers who have made libraries I have used, are mentioned here too. Magister Katarina Kocian has read my thesis very often to find even the last mistake. Without these people it would not have been possible to write this thesis.

Contents

Chapter 1

Introduction

Options are derivative securities. At expiration date the value of an option is exactly determined by an underlying cash instrument. The process of finding the value of an option before expiration date is called option pricing. The history of the theory of option pricing began in 1900 when the French mathematician Louis Bachelier derived an option pricing formula. His formula is based on the assumption that stock prices follow a Brownian motion with zero drift. Since that time, numerous researchers have contributed to the theory. In the year 1973 Fischer Black, Myron Scholes and Robert Merton made a breakthrough in the pricing of options. They have derived a single equation for pricing options, under the assumption of a lognormal distribution of the underlying asset. Still there are some problems with the model's assumption. Empirical evidence (compare with [BCK92]) has shown that the underlying securities do not behave according to that assumption. The probability of large price changes is much higher than it should be possible under the lognormality assumption. Another typical feature of empirical return distributions is called heteroskedasticity, the changing of the variance in time. In practise, many return series show volatility clustering, where bad news lead to a significant increase of the volatility. After some time volatility returns to the old value. The GARCH (Generalized Autoregressive Conditional Heteroskedasticity) model of Tim Bollerslev is an answer to these problems. Jin-Chuan Duan has developed an option pricing model for underlyings following GARCH processes. Still one drawback remains. It is not possible to derive a closed-form equation for option pricing similar to the Black-Scholes formula. Option prices can only be calculated via Monte Carlo simulation, which is computationally expensive and time consuming.

Meanwhile new approaches to solve complex problems evolved in the field of computer science. Many of them have been inspired by the way nature "solves problems". Neural networks are now widely used in different areas. [Hol75] introduced the concept of Genetic Algorithms, which is very successful in the field of Operations Research. [Koz92] enhanced the Genetic Algorithm to the so called Genetic Programming approach, which is applicable in fields as different as electrical engineering and symbolic regression (compare with [K^+03]).

Brokers frequently need to make decisions within seconds. Until recently it was not possible to use the GARCH model for more than a small number of options, because it takes too much time to perform a Monte Carlo simulation. [Han98] used a Neural Network to overcome this problem. [DS01] provide a Markov chain approximation. This master thesis will use Genetic Programming to derive an approximate analytic formula for pricing options when the underlying follows a GARCH process.

The thesis is organized as follows. Chapter 2 provides a brief overview of the concepts of option pricing. Chapter 3 introduces Genetic Programming. Chapter 4 gives an overview of existing approaches. They are all related to this work and are used as a benchmark for the results. Our new approach is presented in chapter 5. It shows also the strategic modus operandi of this work. Chapter 6 discusses implementation issues. This chapter also gives information about the libraries used, which are freely available on the internet. Chapter 7 gives a detailed experimental analysis of the results and a comparison to existing approaches. It includes statistical tests to find out the best configurations. Chapter 8 concludes this master thesis and gives some suggestions for further research.

Chapter 2

Option pricing

This chapter gives a brief overview of option pricing and shows approaches which are used in later chapters. A comprehensive introduction to option pricing can be found in [Hul02]. Some of the more complex mathematical aspects can be found in [Nef00].

2.1 Basic approaches in option pricing

2.1.1 Some definitions and basic models

According to [Nef00], p. 2 "a financial contract is called a *derivative security*, or a contingent claim, if its value at expiration date T is determined exactly by the market price of the underlying cash instrument at time T. Hence, at the time of expiration of the derivative contract, denoted by T, the price F(T) of a derivative asset is completely determined by S(T), the value of the underlying asset. After that date, the security ceases to exist."

The underlying asset can be

- stocks,
- currencies,
- interest rates,
- indexes
- commodities like crude oil, gold and many more.

It is possible to group derivative securities under three general headings:

- Futures and forwards
- Options
- Swaps

A future and a forward contract is an obligation to buy (or sell) an underlying asset at a specified price on a known date. If the specified price is not equal to the market price of the underlying at expiry the holder of the contract makes a loss or a profit.

In contrast to that, an option is the right, but not the obligation to buy (or sell) the underlying asset at a specified price on a specified date. The specified price in the contract is known as the *exercise price* or *strike price*. The specified date in the contract is known as the *expiration date* or *maturity*. If it is a right to buy it is a *call option*, if it is a right to sell it is a *put option* (compare with [Hul02] p. 1 - 15).

Options may be classified by their exercise mode:

American options can be exercised at any time up to the expiration date.

European options can only be exercised on the expiration date itself.

If X is the strike price and S_T is the final price of the underlying asset, the payoff at the expiration time of a European call option is

$$\max(S_T - X, 0). \tag{2.1}$$

This reflects the fact that the option will be exercised if $S_T > X$ and will not be exercised if $S_T < X$. Similarly the payoff at expiration time of a European put option is

$$\max(X - S_T, 0). \tag{2.2}$$

Before expiration, the price of a stock option is affected by six factors ([Hul02]):

- Current stock price.
- Strike price.
- Risk-free interest rate
- Volatility of the stock price, which is the annualized standard deviation.
- Time to expiration.
- Dividends expected during the life of the option.

If the current stock price is high then it is more likely that the stock price at expiration time will be high too. According to equation 2.1 the value of a call option will be higher when the stock price is higher at expiration time. The strike price is not subject to change until the expiration date and influences the value of the option in a direct manner.

The risk-free interest rate affects the price of an option in a less clear-cut way. As interest rates in the economy increase, the expected growth rate of the stock price tends to increase. However, the present value of any future cash flow received by the holder of the option decreases.

As volatility increases, the chance that the stock will do very well or very poorly increases. For the owner of the underlying, these two outcomes tend to offset each other. However the owner of a call benefits from price increases but has limited downside risk in the event of price decreases because he has no obligation to exercise the option. Therefore as volatility increases the value of an option also increases.

The time to expiration influences the value of a call option in two ways. More time until expiration means a higher change of large changes in the underlying price, which increases the price of options. At the same time more interest has to be paid. This decreases the value of an option.

According to [Hul02] p. 170, dividends have the effect of reducing the stock price on the ex-dividend date. This is bad news for the value of call options and good news for the value of put options. The value of a call option is therefore negatively related to the size of any anticipated dividends, and the value of a put option is positively related to the size of any anticipated dividends.

The *moneyness ratio* is defined as S_t/X. A call option is called out-of-the-money if the moneyness ratio is less than 1. If it is worth more than 1, it is called in-the-money. In case it is close to 1 it is called at-the-money.

2.1.2 The Black-Scholes formula

As long as it is possible to determine which process the underlying will follow in the future it is possible to calculate the price of an option before the expiration date too. Because there are countless factors which can influence the price of a stock and these factors are usually not known in advance the process cannot be deterministic, but is stochastic.

Stochastic processes can be classified as discrete-time or continuous-time. A discrete-time stochastic process is one where the value of the variable can change only at certain fixed points in time, whereas a continuous-time stochastic process is one where change can take place at any time (compare with [Hul02] p. 216).

A *Wiener process* or *Brownian motion* is a continuous-time stochastic process. It is a particular type of Markov stochastic process with a mean change of zero and a variance rate of 1.0 per year. A good overview of the properties of a Wiener process may be found in [Nef00] p. 173 - 202.

We are assume that the underlying follows a generalized Wiener process

$$dS = \mu S dt + \sigma S dz \tag{2.3}$$

where S is the price of the underlying, dt is the time between two measure points, μ is the expected return of the underlying, σ is the volatility of the process and z is a standard Wiener process.

This equation implies that stock prices have the lognormal property. This means that the percentage changes (dS/S) of stock prices are normally distributed.

[BS73] found a solution to a stochastic differential equation derived from this process by constructing a locally riskless hedge-portfolio. This leads to the following formula for pricing call options:

$$c = S_0 N(d_1) - Xe^{-rT} N(d_2) \tag{2.4}$$

where

$$d_1 = \frac{\ln(S_0/X) + (r + \sigma^2/2)T}{\sigma\sqrt{T}} \tag{2.5}$$

and

$$d_2 = d_1 - \sigma\sqrt{T} \tag{2.6}$$

The function $N(x)$ is the cumulative standard normal probability distribution function. Furthermore r is the riskless interest rate and T is the time left to maturity. A derivation of this formula can be found in [Nef00].

2.2 Discrete-time stochastic processes and the GARCH model

2.2.1 Discrete-time stochastic processes

Another approach to model the price process of the underlying are discrete-time stochastic processes. In contrast to the generalized Wiener process, only discrete time steps are used. The discretized version of the generalized Wiener process leads to the following equation:

$$\frac{S_t}{S_{t-1}} = \mu - \frac{\sigma^2}{2} + \varepsilon_t \tag{2.7}$$

where S_t is the price of the underlying at time t, μ is its expected return, σ its volatility and ε_t is a normally distributed random variable with mean 0 and variance σ^2. The left-hand side of the equation is defined as the yield (y) of the underlying. With the yield it is possible to calculate S_t from S_{t-1} via the following formula:

$$S_t = S_{t-1}e^y \tag{2.8}$$

As can be seen from equation 2.7 the Black-Scholes model assumes that the probability distribution of the underlying asset at any given future time is lognormal, which is a less than perfect assumption because the probability of high losses and profits is much higher in reality. Therefore the true probability distribution seems to have a higher kurtosis than the normal distribution. Another wrong assumption of the Black-Scholes model is that the volatility of the underlying is constant in time (compare [Hul02] p. 331 - 345).

2.2.2 The GARCH model

The Generalized Autoregressive Conditional Heteroskedasticity (GARCH) model was first introduced by [Bol86]. It is a more flexible variant of the ARCH model proposed

parameter	description
S_t/X	moneyness ratio
r	interest rate
σ^2	unconditional variance
$T-t$	expiration time
$\sqrt{h_t}/\sigma$	relation between conditional and unconditional standard deviation
a_1	GARCH parameter
b_1	GARCH parameter
λ	risk premium

Table 2.1: Input data

by [Eng82]. A good overview of the different types and applications of the GARCH model can be found in [BCK92].

The GARCH model is a discrete-time stochastic process. Every GARCH process consists of two equations. One defines the mean, the other the conditional variance. The yield of a GARCH(1,1) process is defined as

$$y = \frac{S_t}{S_{t-1}} = r + \lambda\sqrt{h_t} - \frac{h_t}{2} + \eta_t \tag{2.9}$$

with

$$h_t = a_0 + a_1\eta_{t-1}^2 + b_1 h_{t-1}, \tag{2.10}$$

where r is the riskless interest rate, S_t is the price of the underlying at time t, λ is the risk premium, h_t is the conditional variance, a_0, a_1, b_1 are GARCH parameters and η_t is a normally distributed random variable with mean 0 and variance h_t ($\eta_t \sim N(0, h_t)$).

The difference between a GARCH process and the discretized Geometric Brownian motion (used in the Black-Scholes model) is that the variance may now change over time. The variance (h_t) depends on the GARCH parameters (a_0, a_1, a_2), the variance from one period before and random disturbances (η_t). Although η_t is normally distributed, the unconditional variance of the whole process is not normally distributed. Therefore, depending on the values of the GARCH parameters, the distribution is not the same as in the Black-Scholes framework (compare with [Han98]).

Usually a GARCH process is estimated by the maximum-likelihood method, where all parameters (a_0, a_1 and b_1) are estimated at the same time. To produce sample time-series Monte Carlo simulation might be used. Until yet it was not possible to find a solution for the differential equations of the GARCH process in general.

2.2.3 Monte Carlo simulation of a GARCH process

The data will be simulated according to [Dua95] and [GS96]. In [Dua95] the concept of locally risk-neutral valuation relationship (LRNVR) was introduced. The LRNVR leads to a transformation of the formula of the yield and the conditional variance, which holds under realistic assumptions about the behavior of investors.

The given input data are shown in table 2.1 and are usually estimated from historical data. The condition $a_1 + b_1 < 1$ must always be satisfied, otherwise the unconditional variance is not defined. The GARCH parameter a_0 is given by the following formula:

$$a_0 = \sigma^2(1 - (1 + \lambda^2)a_1 - b_1) \tag{2.11}$$

The next conditional variance is given by

$$h_{t+1} = a_0 + a_1(\eta_t - \lambda\sqrt{h_t})^2 + b_1 h_t \tag{2.12}$$

where $\eta_t^2 = h_t$ in the first case. Next time the i^{th} instance (this means the i^{th} path of simulation) of $\eta_{t+1,i}$ is constructed with standard normal random numbers $z_{t+1,i}$:

$$\eta_{t+1,i} = \sqrt{h_{t+1}} z_{t+1,i} \tag{2.13}$$

We get the yield with

$$y^*_{t+1,i} = r - \frac{h_{t+1}}{2} + \eta_{t+1,i} \tag{2.14}$$

The new moneyness ratio can be calculated by

$$\frac{S^*_{t+1,i}}{X} = \frac{S_t}{X} e^{y^*_{t+1,i}} \tag{2.15}$$

The yield, error term and conditional standard deviation at the next time points $u = t + 2, ..., T$ are calculated equally:

$$y^*_{u,i} = r - \frac{h_{u,i}}{2} + \eta_{u,i} \tag{2.16}$$

with

$$\eta_{u,i} = \sqrt{h_{u,i}} z_{u,i} \tag{2.17}$$

and

$$h_{u,i} = a_0 + a_1(\eta_{u-1,i} - \lambda\sqrt{h_{u-1,i}})^2 + b_1 h_{u-1,i} \tag{2.18}$$

At the end we get the new moneyness ratio of the share:

$$\frac{S^*_{T,i}}{X} = \frac{S_t}{X} e^{\sum_{u=t+1}^{T} y^*_{u,i}} \tag{2.19}$$

And finally the price of an European call option relative to the strike price K at time t:

$$\frac{C^*_{t,i}}{X} = e^{-r(T-t)} \max(\frac{S^*_{T,i}}{X} - 1, 0) \tag{2.20}$$

To decrease the variance of the different simulation paths variance reduction methods are used. This reduces the time to do the simulation. [BBG97] describes the theoretical fundamentals of this approach. Three methods are described.

The method of antithetic variates is described in [BBG97]. For each set of random numbers (z_{ui}) another set with negative values of these random numbers ($-z_{ui}$) is calculated. An option price is calculated by using the original random numbers (z_{ui}), another by using the negative value of the original random numbers. The variance reduced price is the average of these two prices.

The second is the Empirical Martingale Simulation (EMS) introduced by [DS95]. It is a simple enhancement of the Monte Carlo simulation that ensures that the price estimated satisfies rational option pricing bounds. The new simulation procedure generates the EMS asset prices at a sequence of time points, t_1, t_2, ..., t_m using the following dynamics:

$$S_i^*(t_j, n) = S_0 \frac{Z_i(t_j, n)}{Z_0(t_j, n)} \tag{2.21}$$

where

$$Z_i(t_j, n) = S_i^*(t_{j-1}, n) \frac{S_i(t_j)}{S_i(t_{j-1})} \tag{2.22}$$

$$Z_0(t_j, n) = \frac{1}{n} e^{-rt_j} \sum_{i=1}^{n} Z_i(t_j, n) \tag{2.23}$$

Note that $S_i(t)$ is the $i^t h$ simulated asset path at time t prior to the EMS adjustment.

The last one is called the control variate method which is described in [BBG97]. With the same random numbers used in the calculation of GARCH prices (P^{gsim}) Black-Scholes prices are simulated by using equation 2.8 which leads to a simulated Black-Scholes price (P^{bssim}). But the Black-Scholes price (P^{bsana}) can also be calculated analytically by the Black-Scholes formula given in 2.4. The new GARCH(1,1) price (P^{gcv}) after the control variate correction is

$$P^{gcv} = P^{gsim} - \beta(P^{bssim} - P^{bsana}) \tag{2.24}$$

The β should be chosen to minimize the variance and is therefore:

$$\beta = \frac{Cov[P^{gsim}, P^{bssim}]}{Var[P^{bssim}]} \tag{2.25}$$

Chapter 3

Overview of Genetic Programming

Genetic Programming has been introduced in [Koz92]. It is based on genetic algorithms which were originally described in [Hol75]. These two basic approaches will be introduced in the following sections.

3.1 Genetic algorithms

This section gives a brief overview of genetic algorithms. More information can be found in [BFM97] or in [Mic92]. In biology the evolutionary process results in a selection of the fittest individual in a given environment. The environment might be a specific area, a continent or the whole world. In [Hol75] a general framework for adaptive systems is given. The book shows how these adaptive systems (like the evolutionary process) might be applied to artificial systems. Any problem in adaptation can generally be formulated in genetic terms. Once formulated in those terms, such a problem can often be solved by what we now call the "Genetic algorithm" (compare with [Koz92], p. 17-18).

[Mic92], p. 14 states:

> "The idea behind genetic algorithms is to do what nature does. Let us take rabbits as an example: at any given time there is a population of rabbits. Some of them are faster and smarter than other rabbits. These faster, smarter rabbits are less likely to be eaten by foxes, and therefore more of them survive to do what rabbits do best: make more rabbits. Of course, some of the slower, dumber rabbits will survive just because they are lucky. This surviving population of rabbits starts breeding. The breeding results in a good mixture of rabbit genetic material: some slow rabbits breed with fast rabbits, some fast with fast, some smart rabbits with dumb rabbits and so on. And on the top of that, nature throws in a 'wild hare' every once in while by mutating some of the rabbit genetic material. The resulting baby rabbits will (on average) be faster and smarter than those in the original population because more faster, smarter parents survived the foxes."

3.1.1 Basic terminology

A genetic algorithm works on *individuals* (or *genotypes, structures*), which might be a living organism in nature or a solution to a known problem in an artificial system (=*environment*). Each *individual* is completely described by its constant-size *genome.* This *genome* or *chromosome* may be encoded in bits and bytes, alphanumerical letters or nucleotide bases, like it is done in nature. Chromosomes are also called *strings.* In nature every species carries a certain number of chromosomes (humans for example, have 46 of them). However in artificial problems usually one chromosome is sufficient. Chromosomes are made of units - *genes* (also called *features*, *characters*) - arranged in linear succession. Every gene controls the inheritance of one or several characters (compare with [Mic92]).

Every *individual* has an associated *fitness* value. This *fitness* value describes the capability of an *individual* to survive in the *environment.* Operations designed to mimic the Darwinian principle of reproduction and survival of the fittest are used on a set of individuals (=*population*). A *population* therefore consists of many *individuals* which are in general different, but it can also contain identical *individuals.* An algorithm describes which *individuals* are going to survive where the individuals with better *fitness* will have a competitive advantage.

According to [Mic92] p. 17-18 a genetic algorithm for a particular problem must have the following five components:

- a genetic representation for potential solutions to the problem,
- a way to create an initial population of potential solutions,
- an evaluation function that plays the role of the environment, rating solutions in terms of their fitness,
- genetic operators that alter the composition of children during reproduction,
- values for various parameters that the genetic algorithm uses (population size, probabilities of applying genetic operators, etc.).

Genetic operations which determine a genetic algorithm are

reproduction where one individual can reproduce itself (in real life this means that it is able to live longer than the others).

crossover or *sexual reproduction* where two individuals (*parents*) produce one or more individuals.

mutation where the genome of an individual (and therefore the individual itself too) are changed in a random way.

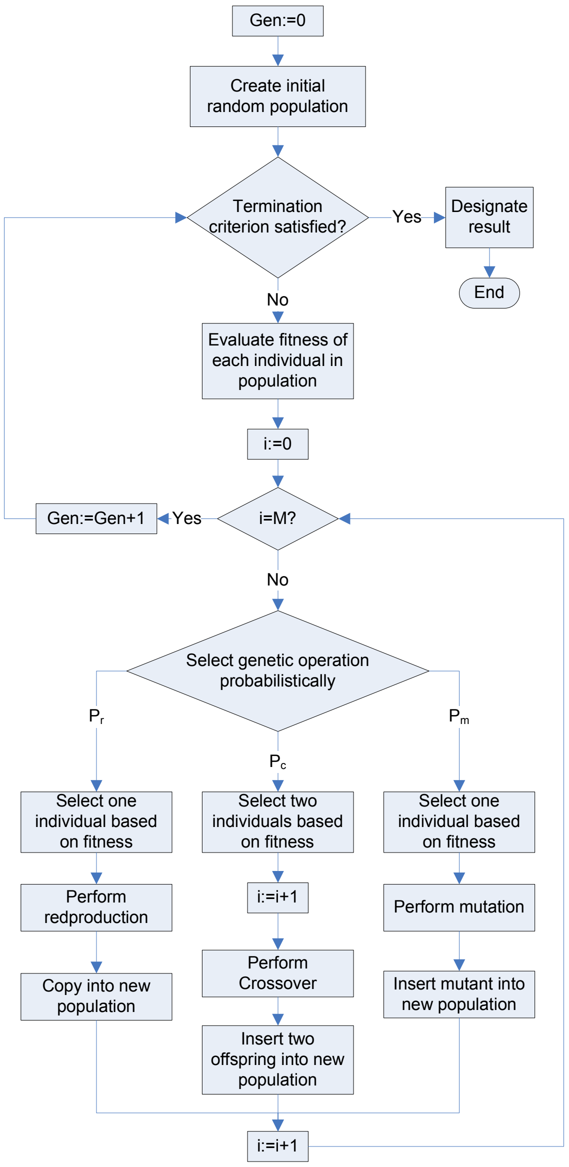
Gen:=0
Create initial random population
Termination criterion satisfied?
Yes
Designate result
End
No
Evaluate fitness of each individual in population
i:=0
Gen:=Gen+1
Yes
i=M?
No
Select genetic operation probabilistically
P_r
P_c
P_m
Select one individual based on fitness
Select two individuals based on fitness
Select one individual based on fitness
Perform redproduction
i:=i+1
Perform mutation
Copy into new population
Perform Crossover
Insert mutant into new population
Insert two offspring into new population
i:=i+1

Figure 3.1: Flowchart of a simple genetic algorithm. Source: [Koz92]

3.1.2 Flowchart of a simple genetic algorithm

In figure 3.1 the basic steps of a genetic algorithm are shown. First the *generation* (=population) counter is set to zero. Then an initial population is generated randomly and the fitness of each individual in the population is examined. If the termination criterion is already satisfied, the genetic algorithm will stop and the result may be used. The termination criterion might be a limit on the number of generations or a specific quality criterion of the individuals. The result is usually the best individual found in the whole algorithm.

To produce a new generation the variable i counts the number of new individuals, which will be set to zero initially. The number of individuals in each population equals M. The next steps are repeated until M new individuals have been created.

The three different genetic operations are chosen randomly. With probability P_r an individual is simply selected, reproduced and copied to the new population. With probability P_m an individual will be mutated and with probability P_c crossover will occur. In this case two individuals are selected, some kind of crossover is performed and the two offspring are copied into the new population.

This is repeated until the new generation is completed. The generation counter is now increased. Then the fitness of each individual is evaluated to find out whether the termination criterion is satisfied.

3.1.3 Additional settings

There are numerous minor variations of the basic genetic algorithm shown in figure 3.1. Some approaches put the mutation operation after crossover and reproduction with some (low) probability. It is not shown how the crossover operation is done exactly. This is often domain dependent, which means that it depends on the specific problem the genetic algorithm is supposed to solve.

Another variation is called the *steady state* framework which is first used in [Rey92] and was originally proposed in [Hol75]. In this case the new population is not produced at once and then replaces the old one, but there exists just one population. The parents are chosen from this population and the generated offsprings are immediately incorporated into the population by replacing one or two existing individuals. The individuals to be replaced can be the worst or randomly chosen ones. An iteration (generations do not exist any more) is considered as completed, once the number of children created by this method is equal to the size of the population. This procedure saves memory and increases the convergence. The disadvantage is that it increases the danger of premature convergence which leads to worse results.

Two selection strategies are important to mention (compare [Fra94]). Possible solutions or chromosomes are assigned a fitness f by the fitness function. In *fitness proportionate selection*, the probability $P^{selection}$ that a specific individual y will be selected is

$$P_y^{selection} = \frac{f_y}{\sum_{x=1}^{N} f_x} \tag{3.1}$$

where N is the number of individuals in the population. While candidate solutions with a lower fitness will be less likely to be selected, there is still a chance that they may be. Contrast this with a less sophisticated selection algorithm, such as truncation selection, which will choose a fixed percentage of the best candidates. With fitness proportionate selection there is a chance some weaker solutions may survive the selection process; this is an advantage, as though a solution may be weak, it may include some components which could prove useful following the recombination process.

Tournament selection chooses a specific number of individuals from the population (for example 10). These individuals are selected randomly with each individual having the same probability to be chosen. Then the best one or best two of these individuals will be selected for the genetic operation. This method is easier to implement and more robust than fitness proportional selection. The disadvantage is that the fitness measure is no longer an absolute value which determines the probability to be chosen. The fitness measure just provides a relative measure for the selection process.

With *demetic grouping* the selection process may also be altered. In this case the whole population is subdivided into a number of groups. These groups undergo their own genetic algorithm and interact only rarely. This allows the *demes* to evolve along separate paths with solutions differing more from each other, which might lead to better individuals at later generations.

3.2 Genetic Programming

Genetic Programming was introduced by [Koz92]. A good introduction can be found in [BNKF98]. It is an extension of the genetic algorithms where the chromosomes are of variable size. The chromosomes now describe hierarchical computer programs encoded in tree-like structures. This leads to additional complexity and some further issues described in the following subsections.

According to [KBAK99], p. 33 the five major preparatory steps for Genetic Programming entail determining

1. the set of terminals (e.g., the actual variables of the problem, zero-argument functions, and random constants, if any) for each branch of the to-be-evolved computer program,
2. the set of primitive functions for each to-be-evolved branch,
3. the fitness measure (or other arrangements for explicitly measuring fitness),
4. the parameters for controlling the run, and
5. the termination criterion and the method of result designation for the run.

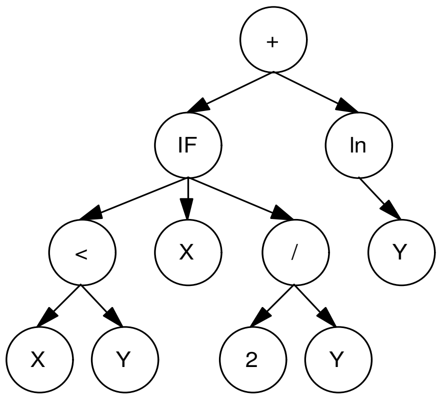

Figure 3.2: An example of a genetic program

3.2.1 Terminal set and function set

An example of a genetic program is given in figure 3.2. With this tree-like representation it is possible to describe every computer program. In the example we have an *IF* statement, some elementary mathematical functions, variables and a constant. The statements may be subdivided into a *function set* and a *terminal set.* The *terminal set* consists of end-points of the tree (=leaves) which do not have any arguments. The variables and constants are subsets of it. The *function set* statements or non-terminals all have arguments (one or more subtrees) and consist of the subsets of arithmetic operations (+,-,*,/), mathematical functions (such as sin, log), Boolean operations, conditional operations (like IF) and functions causing iterations (like do-until loops).

The function and the terminal set must fulfill two properties. The *closure* property requires that each of the functions be able to accept, as its arguments, any value and data type that may possibly be returned by any function and any value that may possibly be assumed by any terminal (compare with [Koz92], p. 81 - 88). The divide function for example does not fulfill the closure property when the second argument is zero. Solutions to this problem are to define an extra value, like "undefined" or the function may return a very high (or very low when maximizing) value. The *sufficiency* property requires that the set of terminals and functions are in principle capable of expressing a solution to the given problem.

3.2.2 Creation of the initial population

The initial population of Genetic Programming is usually created in some random way. Here, we have the problem of variable size chromosomes. The type of creation performed can (according to [Koz92], p. 91 - 94) be one of five types:

Variable: Where a created genetic program can be of a size or structure up to the maximum depth specified for creation. This means that at any point in the tree a function or terminal is arbitrarily chosen. If the maximum size is reached (which need not be the case) a terminal will be selected.

Grow: Where the creation mechanism can only choose functions until the maximum depth is reached when a terminal must be chosen. This causes the size and structure of the genetic program to be the same in all random creations.

Ramped variable: It changes the variable creation type so that the creation mechanism attempts to produce genetic programs with increasing limiting depths up to the maximum depth for creation.

Ramped grow: It changes the grow creation type so that the creation mechanism attempts to produce genetic programs with increasing limiting depths up to the maximum depth for creation.

Ramped half-and-half: This is the creation mechanism used in the majority of Genetic Programming applications. The algorithm permits half the population to be created with ramped variable and the other half to use ramped grow (compare with [Fra94]).

3.2.3 Genetic operations

Reproduction is defined similarly as in the genetic algorithm approach. It just copies the selected individual to the next generation.

Crossover is a little more complicated. Figure 3.3 shows two parents and two possible offsprings. After two parents have been selected, one random point in each parent has to be found. In figure 3.3 these are in both cases the right-hand sides after the root functions. Note that the two parents typically are of unequal size and that the crossover point may also be above a terminal. Then the subtrees are exchanged and the results are the two children for the new generation (compare [Koz92], p. 101 - 105). If accidentally the newly created individuals are bigger than the maximum allowed size, new crossover points have to be chosen and the crossover operation is repeated.

The *mutation* operation introduces random changes in structures of the population. The point where mutation takes place may be chosen randomly like it is done in the crossover operation. According to [Fra94], two types of mutation are meaningful. *Allele mutation* just swaps an n-argument function with another n-argument function or a terminal with another terminal. This operation preserves the shape of an individual. *Shrink mutation* as described in [Fra94] takes the child of a particular gene and moves that child into the position of the parent. This means that genetic programs will shrink. This is a particularly useful property considering how large some genetic programs get as the evolutionary process continues. In [KBAK99], p. 43 - 44, another approach to mutate genetic programs is used. A random point is chosen in the parental program. The subtree rooted at the chosen mutation point is deleted from the program, and a new subtree is randomly grown, using the available functions and terminals in the same manner as trees are grown when creating the initial random population of generation 0. Then the random subtree is implanted at the chosen mutation point. According

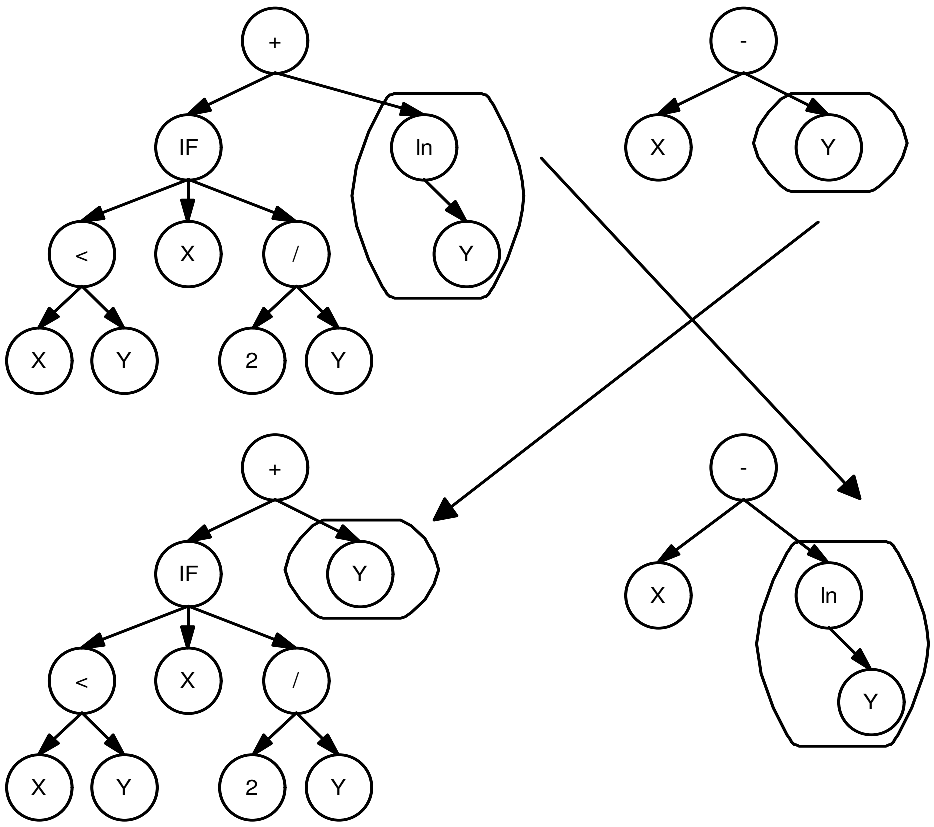

Figure 3.3: An example of a crossover operation

to [KBAK99], p. 44, "the mutation operation is generally sparingly used in Genetic Programming".

An operation which is not defined in genetic algorithms is *editing*. This operation is asexual in that it operates on only one individual. The editing operation applies some domain-specific editing rules. This usually decreases the size of individuals by mathematical simplifications of formulas and similar editing rules.

3.2.4 Automatic defined functions

In [Koz94] the concept of *automatic defined functions* (=ADF) is elaborated in detail. With ADFs the search space may be significantly decreased by using problem domain knowledge. In fact ADFs correspond to the concept of subroutines which may be reused many times by the main program.

Therefore the set of non-terminals will be augmented by some ADFs (=subroutines) in the main tree. These subroutines may have a specific (domain dependent) number of arguments. Each ADF now is a genetic program itself and has a terminal and a non-terminal set, which might be useful to a specific problem.

If a problem can be divided into subproblems then ADFs are useful. The description of each ADF equals the description of the subproblem. In general it is not necessary to have premature knowledge of the subproblem. This can be solved by the Genetic Programming algorithm. The whole problem may then be solved by using the solutions of the ADF. Another advantage is that an ADF might be used in the main genetic program more than once. This increases the efficiency of the genetic program.

If ADFs are used in addition to the five major preparatory steps (compare with chapter 3.2), the architecture of the to-be-evolved program must be determined in some way. For example this can be the number of different ADF's and the number of arguments each ADF has. According to [KBAK99] p. 71 - 74, five methods have been used previously for making the necessary architectural choices. The first four of these methods are manual; the fifth is an automated technique:

1. prospective analysis of the nature of the problem, where human insight is used after analyzing the problem in advance,

2. retrospective analysis of the results of actual runs of similar problems,

3. seemingly sufficient capacity,

4. affordable capacity, which might be the memory-size of a computer, and

5. evolutionary selection of the architecture.

The last method uses the same genetic algorithm to choose the architecture. After generation 0, there is a competition among the existing architectures during the course of the run. This means that individuals with different architectures exist in every population. A whole book [KBAK99] describes in detail all the problems which arise from this approach and how new architectures may emerge and bad architectures may disappear.

3.2.5 Bloat control

When evolutionary computation uses arbitrarily-sized representations, often the evolutionary process progresses not only towards fitter individuals, but to dramatically larger individuals. This rapid increase in size, known as bloat, can hinder the evolutionary mechanism itself and can slow down successive generations to a point where further progress is not feasible anymore (compare with [PL04]). This is especially true when Genetic Programming gets out of memory.

According to [PL04], the most common approach to bloat control is establishing hard limits on size or depth, primarily because this approach was popularized by early work in Genetic Programming [Koz92]. If a newly created child is deeper than a specific number, it is rejected and a new child will be created with lower depth. *Depth limiting* has proven a surprisingly successful method and usually all other methods are combined with some restrictions on tree depth as well.

Another straightforward approach is called *parsimony pressure*. Parsimony pressure has often been parametric, meaning that it considers the actual value of size and fitness together in a parametric statistical model for selection. This may easily be done by adding a penalty to the fitness function which is proportional to the size of the individual. Care has to be taken of the penalty factor, because this influences Genetic Programming in a high manner. A penalty factor that is too high leads to small (in size) but poor (in quality) results.

[PL04] suggests a new approach which is called *death by size*. Death by size is intended for methods such as steady-state evolution which requires a procedure for marking individuals for death. The bloat control approach uses fitness to select individuals for breeding, as usual, but when selecting an individual for death, selection is done by size (preferring to kill larger individuals).

3.2.6 Control parameters

[Koz92] has defined 19 control parameters which determine a Genetic Programming algorithm. These control parameters include such important variables as the population size, the maximum number of generations and some probabilistic values. In table 3.1 these parameters are shown with the default values proposed by [Koz92]. Of course in practice the values of the control parameters should be domain-dependent.

Parameter	Default value
Population size	500
Maximum number G of generations to be run	51
Probability of crossover	90 %
Probability of reproduction	10 %
Probability of choosing internal points for crossover	10 %
Maximum size for S-expressions created during the run	17
Maximum size for initial random S-expressions	6
Probability of mutation	0 %
Probability of permutation	0 %
Frequency of editing	0
Probability of encapsulation	0 %
Condition for decimation	NIL
Decimation target percentage	0 %
Generative method for initial random population	ramped half-and-half
Basic selection method	fitness proportionate
Spousal selection method	fitness proportionate
Adjusted fitness	used
Over selection	not used
Elitist strategy	not used

Table 3.1: Control parameters

Chapter 4

A survey of existing approaches in option pricing

This chapter gives a brief overview of existing approaches in option pricing. [Han97] and [DS95] use two different methods to price options, where the underlying follows a GARCH process. [Keb99] is the first attempt to use Genetic Programming in option pricing at all. He assumes that the underlying follows a Wiener process (compare with chapter 2.1.2). These approaches will also be used later to assess the quality of the results of the new approach. The new approach will use Genetic Programming to price options following a GARCH process (compare with chapter 5).

4.1 Neural networks

The first approach to use neural networks for pricing options according to a GARCH model was introduced by [Han98] (some results may also be found in [Han97]). In this doctoral thesis a huge variation of different models and options have been estimated by neural networks. It ranges from European to Asian options, from the Black-Scholes framework to the GARCH framework and even empirical data have been used to train the neural networks. At the end an adaptive non-linear model has been proposed, which is built upon a hybrid approach of Black-Scholes prices and the daily deviation between these prices and the market prices.

The network topology used in [Han97] is feedforward, i.e. loop-free. Three layers have been used, where each layer is connected to the next one fully (i.e. each neuron of a specific layer is connected to every neuron of the next layer). The first layer is called the input layer. This layer has the number of input parameters as the number of neurons. Each neuron is therefore assigned to a specific input parameter. The second layer is called the hidden layer. This layer is tested with different numbers of neurons for each problem. It ranges between 4 and 40 neurons with step size 4. The third layer is called the output layer. It consists of only one neuron, because the output should only lead to one result: The price of the given option.

The activation function of the neurons in the hidden layer is the hyperbolic tangent function:

$$\psi(x) = \frac{1-e^{-x}}{1+e^{-x}}, \tag{4.1}$$

where x is the sum of the weighted values of all input neurons. The output neuron computes the sum of all values from the hidden layer using its own weights and it applies the identity function ($id(x) = x$) as activation function. This leads to the following mathematical representation of the whole neural network:

$$o(x,w) = w_0 + \sum_{j=1}^{J} w_j \frac{1-e^{-\sum_{i=0}^{I} w_{ij}x_i}}{1+e^{-\sum_{i=0}^{I} w_{ij}x_i}} \tag{4.2}$$

where $o(x,w)$ is the output (the result) of the neural network, I is the number of input parameters and J is the number of hidden neurons. x_i is therefore a specific input parameter, except x_0 which is a constant. All w's are the weights of the connections between two neurons.

The determination of the weights (w) is called learning. In [Han97] the back propagation algorithm and the Broyden-Fletcher-Goldfarb-Shanno one-step memoryless quasi-Newton method is used for learning. In all cases the second method leads to better results. Both learning methods are supervised, which means that they use data (historical or model driven) to determine the weights. 1000, 2500, 5000 and 10000 data points are used for each learning algorithm.

In the first step the neural network is trained according to the Black-Scholes model and then the results are compared to the calculated values (with the Black-Scholes formula). The best result gives a root mean squared error (RMSE) of $1.99 * 10^{-4}$ where the (RMSE) is defined as follows:

$$RMSE = \sqrt{\frac{1}{N}\sum_{n=1}^{N} e(n)^2} \tag{4.3}$$

with N the number of data points and $e(n)$ the network error.

In the second step European call options are estimated under the GARCH model. The simulation of the GARCH prices is done exactly as described in chapter 2.2.3. The input data are uniformly distributed in the ranges listed in table 4.1. The best results give a RMSE of $5.63 * 10^{-4}$.

In both cases a short analysis of the deviation related to some input data ranges is given. Noticeable is the higher deviation for at-the-money options. The deviation is also significantly higher for short-maturity options. Additionally in the GARCH case the deviation is higher at long maturities.

4.2 Markov chain approximation

[DS01] propose a Markov chain approximation to price options. The stock prices and the volatilities are assumed to be discrete and elements of a fixed set. This fixed

parameter	description	range
S_t/X	moneyness ratio	[0.7,1.3]
r	annualized risk free interest rate	[0,0.1]
σ^2	annualized unconditional variance	[0.01,0.16]
$T-t$	expiration time in days	[1,30]
$\sqrt{h_t}/\sigma$	relation between conditional and unconditional standard deviation	[0.8,1.2]
a_1	GARCH parameter	[0,1]
b_1	GARCH parameter	[0,1]
λ	risk premium	[0,0.001]

Table 4.1: Input data ranges in [Han98]

partitioning of the state space simplifies the task of option pricing to a sequence of standard matrix operations. Under a Markov chain representation, the conditional expected value is simply a product of two components. For European options, the first component is the transition probability matrix raised to a power equal to the maturity of the option (measured in terms of the basic transition period), and the second is the payoff vector associated with the option.

The time step is fixed to one day. A $mn \times 1$ vector $\overline{V}(t)$ contains the approximate option values at time t for all possible states. This means that the stock price can only have m different values while the volatilities can take n different values. The transition probability matrix is given by the conditional distribution function. Additionally the option values at each time step are adjusted to

$$S_t^* = e^{-(r-(1/2)h^*)t} S_t \tag{4.4}$$

to avoid a trend in time of the option prices. Additional information about the problems of implementing American options by a Markov chain approximation can be found in [DS01].

Four different types of options are evaluated. The first two are European put options and American put options in the Black-Scholes framework. The others are European put options and American put options in the GARCH framework. The GARCH option pricing model used in the paper is the NGARCH(1,1) model. This model has an additional term to capture the leverage effect (by the leverage parameter θ).

The European put prices in the Black-Scholes framework are compared with the Black-Scholes prices. The parameters like the interest rate and the GARCH parameters are set to realistic values (see table 4.2). Three values for the maturity (1, 3 and 9 months) and three values for the moneyness ratio (1.1, 1.0 and 0.9) are used. In the Black-Scholes framework, volatility is assumed to be constant, therefore $n = 1$, while m is simulated in the range between 11 and 501. The results are good, they only vary in the range of 10^{-4} for $m = 501$.

parameter	describtion	value
S_0	initial price of the option	50
r	annualized interest rate	0.05
β_0	GARCH parameter	0.00001
β_1	GARCH parameter	0.8
β_2	GARCH parameter	0.1
θ	leverage factor	0.3
λ	risk premium	0.2

Table 4.2: Parameter values in [DS01]

parameter	description	value
S_0	initial price of the option	[10,100]
r	annualized interest rate	[3%,7%]
σ	annualized volatility	[5%,50%]
α	moneyness ratio	[0.9,1.1]
ΔT	maturity of the option	[0,1] years

Table 4.3: Parameter range in [Keb99]

European put prices under the GARCH framework are compared to prices calculated by Monte carlo simulation. Realistic values for the parameters are chosen and listed in table 4.2. The values for the moneyness ratio and the maturity are chosen as before. The values of n are in the range between 25 and 51, while m ranges from 25 up to 357. When $m = 357$ and $n = 51$ the results vary dependent on the maturity (higher maturity means less accuracy). At a maturity of 1 month the error is at worst $6 * 10^{-3}$ and at 3 and 9 months around $12 * 10^{-3}$.

4.3 Genetic Programming

The first implementation of Genetic Programming for option pricing is [Keb99]. He uses this approach to determine a formula for American put options. In this case the prices of the options are calculated by the finite difference method (an introduction can be found in [Hul02], p. 418 - 426). The ranges of the parameters are given in table 4.3.

As terminals he uses the variables S_0, X, r, T, σ, α and the constants π and e. As functions he uses the mathematical functions $+$, $-$, $*$, $/$, $\sqrt{x}$, $\ln(x)$, x^2, x^y, the distribution function of the standard normal distribution $\Phi(x)$, the logical functions $<$, $\leq$, $=$, $>$, $\geq$ and the functions with side effects IF, THEN and ELSE.

The size of the population is fixed with only 50 individuals, the crossover and mutation probability were $p_c = 0.9$ and $p_m = 0.01$ respectively. The number of generations is set to 20000. The fitness function is defined as

$$f_r(c) = \sum_{i=1}^{n} \Delta(A_i, A_i^S) \tag{4.5}$$

where A_i refers to the value calculated by the genetic program and A_i^S refers to the real value.

The results are good compared to other numerical approaches to find prices for this specific type of option. The derivation of the evaluation equation yields - as a by-product - an expression for the killing price. This gives the pricing equation a nice economic interpretation. Usually the equation derived via Genetic Programming is empirically good, but it is not possible to analyze them economically.

A disadvantage of this approach is that usually the underlying of the option doesn't follow a Wiener process, but more likely a GARCH process. Still the result of the work shows that it is possible to use Genetic Programming in the area of option pricing and that even American options can be priced via this method.

Chapter 5

Strategic decisions of the new approach

This chapter describes all the strategic decisions and approaches used in this master thesis. The result should be an analytic formula that calculates prices of European options when the underlying follows a GARCH(1,1) process. Therefore, in a first step, sample data points have to be generated. Then Genetic Programming searches for a formula (which is a subset of a program) which fits best to the sample data points. This task is called symbolic regression (compare with [Koz92]).

5.1 GARCH process

The input data are used similar to [Han98] and are shown in table 5.1. All input data are uniformly distributed in the given ranges except S_t/X has a normal distribution with mean 1 and standard deviation 0.1.

Option prices are then simulated using the Monte Carlo approach described in

parameter	description	range
S_t/X	moneyness ratio	N(1,0.1)
r	annualized risk free interest rate	[0,0.1]
σ^2	annualized unconditional variance	[0.01,1.00]
$T-t$	expiration time (days)	[1,90]
$\sqrt{h_t}/\sigma$	relation between conditional and unconditional standard deviation	[0.8,1.2]
a_1	GARCH parameter	[0,1]
b_1	GARCH parameter	[0,1]
λ	risk premium	[0,0.001]

Table 5.1: Input data and ranges

chapter 2.2.3. In the data set all values are transformed from annually to daily, with the assumption of 260 trading days a year. The number of simulation paths is set to 10000 to get values with a low variance. Two samples are produced, one with 1000 data points and one with 10000 data points. Both are used in my analysis, the first one for the derivation of option prices via Genetic Programming, and the second one to evaluate the results. On my computer (AMD Athlon XP 2200+ CPU with 512 MByte DDRAM) it took around one hour for the 1000 data points and around ten hours for the 10000 data points to be generated via Monte Carlo simulation.

5.2 Genetic Programming

5.2.1 Functions and terminals

Table 5.2 shows all the terminals used in the Genetic Programming algorithm. The terminal set consists of constants and variables. The constants 0, 1 and 2 are added to get some basic numbers. The Genetic Programming algorithm can generate all numbers of the set of rational numbers from these constants via simple functions. The constant 2 is selected to allow for easy generation of the square and the square root from the *power* function. The constants π and e increase the numbers the Genetic Programming algorithm can create to some irrational numbers. They also support the creation of some special functions from basic functions (e.g. e^x).

The remaining terminals correspond to the input parameters of section 5.1. Assuming that these variables have a large impact on the value of the option, all variables must be part of a function (genetic program) which calculates the price of the option.

Table 5.3 shows all the functions used in the Genetic Programming algorithm. They consist of the four basic functions addition, subtraction, division and multiplication. The natural logarithm and the power function are also used to get functions like e^x and many more. The maximum and the minimum function seem to be important because of equations 2.1 and 2.2.

Due to the utilization in the Black-Scholes formula and in [Keb99], a distribution function seems to be useful too. The normal distribution function, the Student-t distribution function and the paretian stable distribution function, respectively, are therefore added. The paretian stable distribution function was calculated according to [Nol97].

5.2.2 Fitness function and bloat control

The overall task is to find an equation that approximates as good as possible the price of a call option depending on the input data. This is done via Genetic Programming as described in [Koz92]. The fitness function will be the root mean square error (RMSE) between the result of the genetic program and the real result (see also equation 4.3).

One major task in Genetic Programming is to prevent the resulting equations from being too large in size. The concept is called bloat control (compare with [PL04]). Three methods which are all described in [PL04] are used in this master thesis.

Symbol	description	type
0	zero	constant
1	one	constant
2	two	constant
π	PI	constant
e	Euler's constant	constant
S_t/X	moneyness ratio	variable
r	annualized risk free interest rate	variable
σ^2	annualized unconditional variance	variable
$T-t$	expiration time (days)	variable
$\sqrt{h_t}/\sigma$	relation between conditional and unconditional standard deviation	variable
a_1	GARCH parameter	variable
b_1	GARCH parameter	variable
λ	risk premium	variable

Table 5.2: Terminal set

name of the function	description	number of arguments
addition	$x+y$	2
subtraction	$x-y$	2
division	x/y	2
multiplication	xy	2
natural logarithm	$\ln y$	1
power function	x^y	2
maximum	$\max(x;y)$	2
minimum	$\min(x;y)$	2
normal distribution function	$N(x)$	1
student distribution function	$S(t,x)$	2
paretian stable distribution function	$P(\alpha,\beta,x)$	3

Table 5.3: Function set

name of the approach	number of genetic program trees	equation assumed
standard	1	no
standard with ADF	2	no
hybrid without ADFs	$2-6$	yes
hybrid with ADFs	$3-9$	yes

Table 5.4: Different approaches used

The first is called *depth limiting* method, where the resulting tree is restricted to certain depth limit. In this case it will be set to a small number between 7 and 10. The other two approaches use a depth limit too, but it will be relaxed to 20.

The second approach is called *parsimony pressure*. In this approach, the fitness function penalties bigger trees by the following equation:

$$Fitness = StandardFitness + \frac{Length}{Penalty} \tag{5.1}$$

where *Length* means the number of nodes in a given genetic program and *Penalty* is a constant penalty factor. The higher *Penalty* is, the lower bigger trees will worsen the Fitness. Values between 50000 and 100000 have shown to lead to good results.

The third method is introduced in [PL04] and is called *death by size*. Here bigger individuals are selected for death randomly at some time in the program.

5.2.3 Population size, number of generations and mutation

Two different major configurations are used. The configuration *without mutation* is proposed by [Koz92]. Here we have a huge population size (like 40000) and no mutation at all. A small number of generations (like 50) is assumed to be sufficient to build the best fitted solution.

The configuration *with mutation* uses only a small population size (like 4000) but a much higher number of generations (like 500). To ensure that genetic material cannot be lost forever, mutation is used in this case. This approach was used in an even more extreme way by [Keb99].

5.2.4 Automatic defined functions and hybrid approaches

Four different approaches are selected. They differ in the number of automatic defined functions (ADFs) and in the type of equations assumed. In table 5.4 they are listed.

The *standard* approach uses just one genetic program tree, where all functions of the function set from table 5.3 might be used. (Usually just one of the three distribution functions is used.) Also all terminals from the terminal set from table 5.2 are used. Therefore the task of the Genetic Programming algorithm is to find a single equation which fits best to the given data set, without the use of any pre-defined knowledge and automatic defined functions.

The *standard with ADF* approach uses two genetic program trees at the same time. In addition to the standard approach the main tree can also call a second genetic tree with two arguments. The second genetic tree uses all functions from the function set except for the distribution functions. As terminals it uses only the two arguments which are calculated at run-time, depending on the kind of arguments it gets from the main program. This corresponds exactly to the concept of automatic defined functions with one ADF and two arguments as described in chapter 3.2.4.

The two hybrid approaches assume an equation similar to the Black-Scholes function, which has the following form:

$$C_T/X = S_t/X cdf(f_0(x), ...) - e^{-r(T-t)} cdf(f_1(x), ...) \tag{5.2}$$

Depending on the type of the cumulated density function (*cdf*) it may have one (normal), two (student-t) or three (stable paretian) arguments, where each argument is a genetic program tree (f_x). In the *hybrid without ADFs* approach, the $2-6$ genetic program trees are not used with ADFs and are therefore solved separately. In the *hybrid with ADFs* approach two genetic program trees, which represent the same argument in the two cdf's, have one additional genetic program tree. This additional genetic program tree might be called from the two genetic program trees without arguments and is therefore an ADF.

Chapter 6

Implementation details

This chapter provides an overview of the implementation details of the genetic program. It shows which libraries are used and how they are used. Furthermore, all classes are described and finally a UML (Unified Modeling Language) diagram gives some insight into the relationships between the classes. It does not provide a full description of the source code.

6.1 Libraries

The whole program is written in C++ and should correspond to the ANSI standard (compare with [Str98]). It should therefore be platform independent. In fact, I was able to run it under Microsoft Windows 2000 and SUSE LINUX 9.0 without changing any line of code. The CD you can find attached to this thesis reflects this by offering a standard make-file for Linux and at the same time a configuration file for Microsoft Visual C++ 6.0.

Luckily, it was not necessary to implement the whole program from scratch. I take advantage of the possibility to use several libraries which are available for free from the internet. The next few chapters give a short overview of all libraries I have used with www-links and some additional information.

6.1.1 The Genetic Programming kernel

The first Genetic Programming library in C++ was implemented by [Sin94]. It was not really object-orientated and pretty hard to use. Therefore, in [Fra94] a better approach was started. Later it was modified by [Wei97][1] and is now a save and widely usable tool for a wide range of problems that make use of the Genetic Programming approach.

Figure 6.1 shows the class hierarchy of this library. It is according to the *UML* Version 1.5 specification, which can be found in [Obj03]. Two different abstract classes are defined. The class *GPObject* is the base class of all classes used in this library.

[1]The library can be found at http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/weinbenner/gp.html

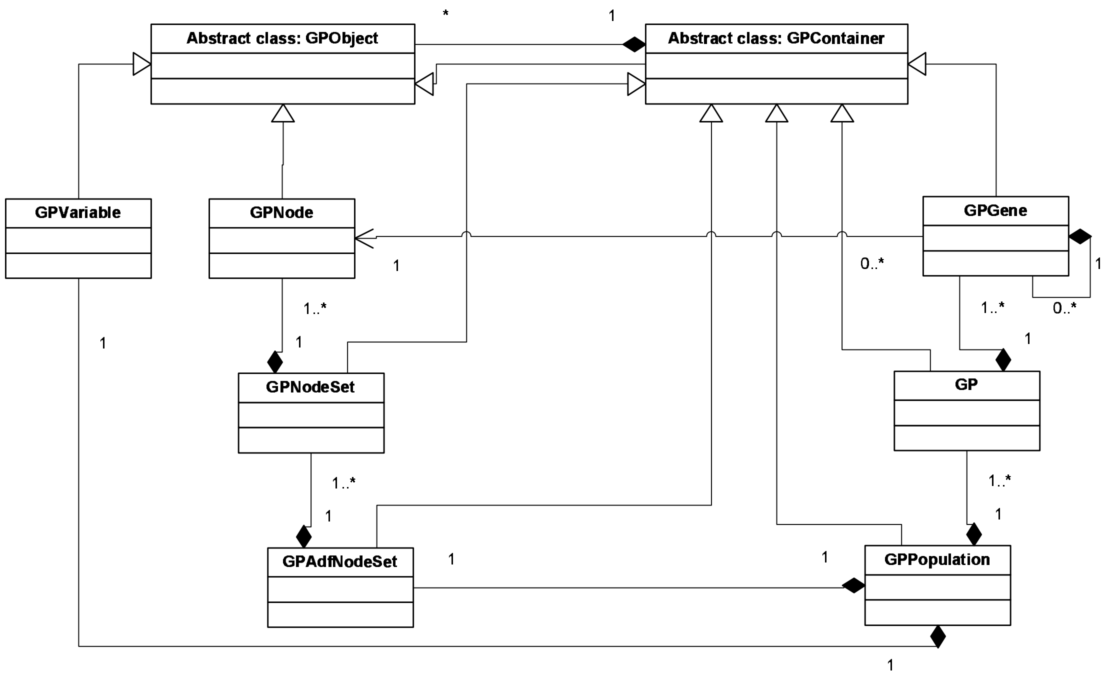

Figure 6.1: UML diagram of the Genetic Programming kernel. Source: [Wei97]

This means that all classes are derived from GPObject. Another abstract class is *GPContainer*. This class manages an array of objects of type GPObject. It provides all necessary methods to handle its objects.

GPNode is a class derived from GPObject which represents a node that can be a terminal or a function, respectively. Each genetic tree has its own function and terminal set. For this purpose, the class *GPNodeSet* is introduced and serves as a container to hold all the different nodes. A genetic program consists of the main tree and the ADF trees. Each tree type can have different functions and terminals. This allows the user to introduce a priori knowledge of the task to be performed by the genetic program (compare with chapter 3.2.4). The container class *GPAdfNodeSet* is used to collect the node sets for each ADF and the main tree, respectively. If the GPAdfNodeSet object has only one GPNodeSet object, no ADFs were defined (compare with [Wei97]).

A *GPPopulation* object is a container that contains all the genetic programs of a population. It also has one object of type GPAdfNodeSet to have the information of all the different nodes it might use for creation. One genetic program is represented by the class *GP*. For each ADF and the main program, a GP contains one object of type *GPGene* which represents the root of a genetic program tree. Genetic Programming uses this tree structure to internally store genetic programs (compare with chapter 3.2.1). A tree consists of genes, each of which can be the parent gene for one or more children. A gene is also a container, so the class GPContainer is used as a base class. A gene object has only one component: a pointer to an object of class GPNode. By referencing a node object, GPGene knows which type of function it represents (compare with [Wei97]).

Parameter	Possible range
Population size	[1...MaxInt]
Number of generations	[0...MaxInt]
Crossover probability	[0.0...100.0] %
Creation probability	[0.0...100.0] %
Creation type	[0...5]
Maximum depth for creation	[2...MaxInt]
Maximum depth for crossover	[MaximumDepthForCreation...MaxInt]
Selection type	[0...1]
Tournament size	[1...MaxInt]
Demetic grouping	[0...1]
Deme size	[0...PopulationSize]
Demetic migration probability	[0.0...100.0]
Swap mutation probability	[0.0...100.0]
Shrink mutation probability	[0.0...100.0]
Add best to new population	[0..1]
Steady state	[0..1]

Table 6.1: Properties of GPVariable. Source: [Wei97]

The *GPVariable* class is used to control the behavior of several aspects of Genetic Programming. Each GPPopulation object has an object of this class as a member. Table 6.1 shows all the variables used by this library. The creation type is the same as defined in chapter 3.2.2, except that there is an additional one, which is a user defined creation type. The selection type may be probabilistic selection or tournament selection, respectively. "Demetic grouping" and "Add best to population" may be switched on (when equals one) or off (when equals zero).

6.1.2 A random number generator library

Newran02B[2] is a C++ library for generating sequences of random numbers from a wide variety of distributions. It is particularly appropriate for the situation where sequences of identically distributed random numbers are required since the set up time for each type of distribution is relatively long, but the generation of a new random number is fast. The library includes classes for generating random numbers from a number of distributions and is easily extended to be able to generate random numbers from almost any of the standard distributions (compare with [Dav02]).

Figure 6.2 shows the static UML structure of the Newran library. *Random* is a class used by all other classes as base class at least indirectly. It generates uniformly distributed random numbers by the Lewis-Goodman-Miller algorithm with Marsaglia

[2]The library can be found at http://www.robertnz.net

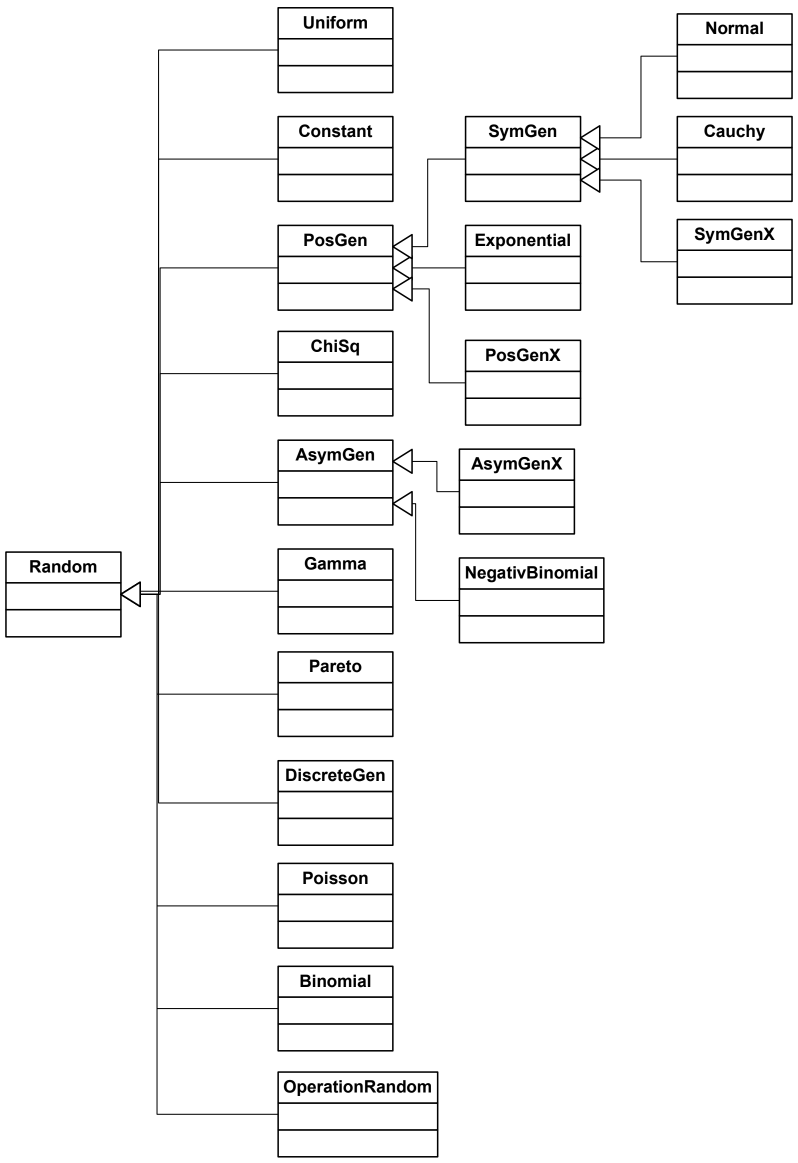

Figure 6.2: UML diagram of Newran02B. Source: [Dav02]

Name	Description
normal_cdf	Returns the normal distribution function
students_cdf	Returns the Student t distribution function
gsl_integration_qag	Numerical integration

Table 6.2: Some functions of the GNU Scientific Library

mixing. Via the static method Set, the random generator is initialized with a real number between zero and one.

It is possible to instance one of the distribution classes to get random numbers via the *Next* method. I have only used the *Uniform* and the *Normal* class to create the random numbers for the Monte Carlo simulation and to get random distributed input data, respectively.

All distribution classes (like Uniform, Pareto, Normal, Cauchy, ...) provide random numbers with different distributions. They usually take as input the uniformly distributed random numbers created by the Random base class. Therefore, they just have to provide an inverse distribution function to generate the random numbers. *OperationRandom* is a synonym for a whole range of classes which provide some functions to existing distribution classes and may also have subclasses. Details can be found in [Dav02].

6.1.3 GNU Scientific Library

The GNU Scientific Library[3] (gsl) is a collection of numerical routines for scientific computing. A complete documentation can be found in [G+04] [4].

The library files *specfns.h* and *gsl_integration.h* contains some globally defined functions which are used in the program. They are listed in table 6.2.

The *normal_cdf* function returns the normal distribution function and the *sutdents_cdf* returns the Student-t distribution function.

The *gsl_integration_qag* implements a numerical integration. This function is used to calculate the distribution function of the paretian stable distribution function. The QAG algorithm is a simple adaptive integration procedure. The integration region is divided into subintervals, and on each iteration the subinterval with the largest estimated error is bisected. This reduces the overall error rapidly, as the subintervals become concentrated around local difficulties in the integrand.

The algorithm is based on Gauss-Kronrod rules. A Gauss-Kronrod rule begins with a classical Gaussian quadrature rule of order m. This is extended with additional points between each of the abscissas to give a higher order Kronrod rule of order $2m+1$. The Kronrod rule is efficient because it reuses existing function evaluations from the

[3]The current stable version is available from ftp.gnu.org in the directory /pub/gnu. The project homepage is http://www.gnu.org/software/gsl/.

[4]This book is also online available at http://www.gnu.org/software/gsl/manual/

Name	Description
stable_cdf	Returns the Stable distribution function
gsl_integration	Gives an easy accessible interface to the gsl integration functions
simpson	Returns the value of a bounded integral according to the simpson algorithm
trapez	Returns the value of a bounded integral according to the trapezoid algorithm
adapt	Returns the value of a bounded integral according to an adaptive algorithm

Table 6.3: Additional functions

Gaussian rule. The higher order Kronrod rule is used as the best approximation to the integral, and the difference between the two rules is used as an estimate of the error in the approximation (compare with [G^+04]).

6.2 New classes and functions

6.2.1 Additional functions

Table 6.3 shows the additionally programmed functions. They are included in the library headers integration.h and specfns.h.

The distribution function of the stable distribution is written according to the numerical approximation in [Nol97]. This is necessary, because there are no analytic formulas for most stable densities and distribution functions. This approximation uses integral formulas for the density and distribution function according to [Zol86]. Some problems emerge when α is close to 1, then α is set to one. I assume that this is a good approximation.

6.2.2 GPOPdata

To give an easy access to the data and to create the data in a unified way, I have programmed the class *GPOPData* and its subclass. The structure is shown in figure 6.3. GPOPdata is the base class and provides all the basic functions necessary. The methods *save* and *load* provide access to a file, which is in ASCII text style. It is a character separated value (csv) file where all the data are separated by tabs. The methods *value* and *varname* provide access to a value or the name of the variable, respectively. *Size_vars* and *Size_data* return the size of the data matrix.

The method *create* makes the data from scratch. This is the only method which differs in the *GPOPgarch* class. In this case it produces random data according to a GARCH process. These data can be stored and read by a GPOPdata class.

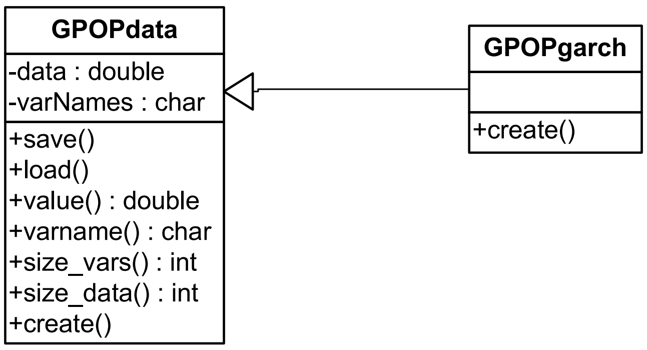

Figure 6.3: UML diagram of GPOPdata

Parameter	Possible range
Bloat method	[0...2]
Penalty	[0...MaxInt]
Normal	[0...1]
Student	[0...1]
Stable	[0...1]
Infofilename	String
Datafilename	String

Table 6.4: Properties of GPVariable. Source: [Wei97]

6.2.3 MyGPVariables

This class is similar to the GPVariables class of the *Genetic Programming kernel* (compare with chapter 6.1.1). It adds some additional parameters to the program. In table 6.4, all the additional parameters are listed. If the *bloat method* parameter (compare with chapter 3.2.5) is set to 0, no bloat control (except depth limiting) will be enforced. If the parameter is set to 1, parsimony pressure, and if it is set to 2, death by size will be used. The *penalty* parameter is only necessary when parsimony pressure is used and leads to a worse fitness (compare with chapter 5.2.2).

The parameters normal, student and stable add the appropriate distribution function to the function set, when they are set to 1 instead of 0. Additionally the name of the resulting file (Infofilename) and the name of the data source (Datafilename) can be given, too. If not given, the resulting files will have the name “result” and the filename will be “data.in”. All these parameters are determined in the initialization file.

6.2.4 MyGene

The class *MyGene* is derived from the *GPGene* class of the *Genetic Programming kernel*. Table 6.5 shows all the methods which have to be rewritten. The *printOn* method calls the *printMathStyle* method and the *printTexStyle* method, respectively

Methods	Description
printOn	Output function
printMathStyle	Output function to ASCII style
printTeXStyle	Output function to TeX style
evaluate	Evaluates the gene

Table 6.5: Methods of MyGene

Methods	Description
printOn	Output function
evaluate	Evaluates the genetic program

Table 6.6: Methods of MyGP

depending on a global variable. The *printTeXStyle* method prints a Gene according to the TeX style and the *printMathStyle* according to a normal math style with brackets.

One of the core functions in Genetic Programming is the *evaluate* method. It calculates the result of a single gene. It has to provide values for the whole function set defined in the genetic program. The value of variables and constants used in the genetic program has to be returned, too. If ADFs (compare with chapter 3.2.4) exist, it has to provide the arguments of the ADFs.

6.2.5 MyGP

The class *MyGP* is derived from the *GP* class of the *Genetic Programming kernel.* Table 6.6 shows all the methods which have to be rewritten. The *printOn* method outputs all the necessary information for TeX style output files. The *evaluation* method calculates the fitness of a genetic program. It calculates the root of the average of the squared errors of all data points (compare with chapter 5.2.2).

6.2.6 MyPopulation

The class *MyPopulation* is derived from the *GPPopulation* class of the *Genetic Programming kernel.* Table 6.7 shows all the methods which have to be rewritten. The *tournamentSelection* method has to be rewritten to provide death by size as bloat control method.

6.2.7 Executables

According to chapter 5, four different strategies to solve the problem are used. This leads to four different executables (files which are executed by the operating system)

Methods	Description
tournamentSelection	Rewrites the selection strategy

Table 6.7: Methods of MyPopulation

Executable	Description
gpop	Simple Genetic Programming
gpop2	Simple Genetic Programming with pre-build function
adf	Genetic Programming with one ADF
adf2	Genetic Programming with ADFs and pre-build function
proddata	Executable to make file with the GARCH-data

Table 6.8: Executables

for Genetic Programming. They only differ in the type of function which are predetermined for evaluation and in the number and type of ADFs used. Table 6.8 shows the different executables which are produced and what they are supposed to do. They also correspond to the four different approaches from table 5.4.

Proddata is the executable file which simulates the GARCH process to get the sample data points (compare with chapter 6.2.2).

6.3 Overview UML diagram

Figure 6.4 shows a UML diagram of the whole program. The GNU library on the right hand side is globally defined and is used by various parts of the program. The two utilities are the executables and they actually produce output. The output is written to a file, which a user can specify via the initialization file. The name of the initialization file is given as a parameter to the program.

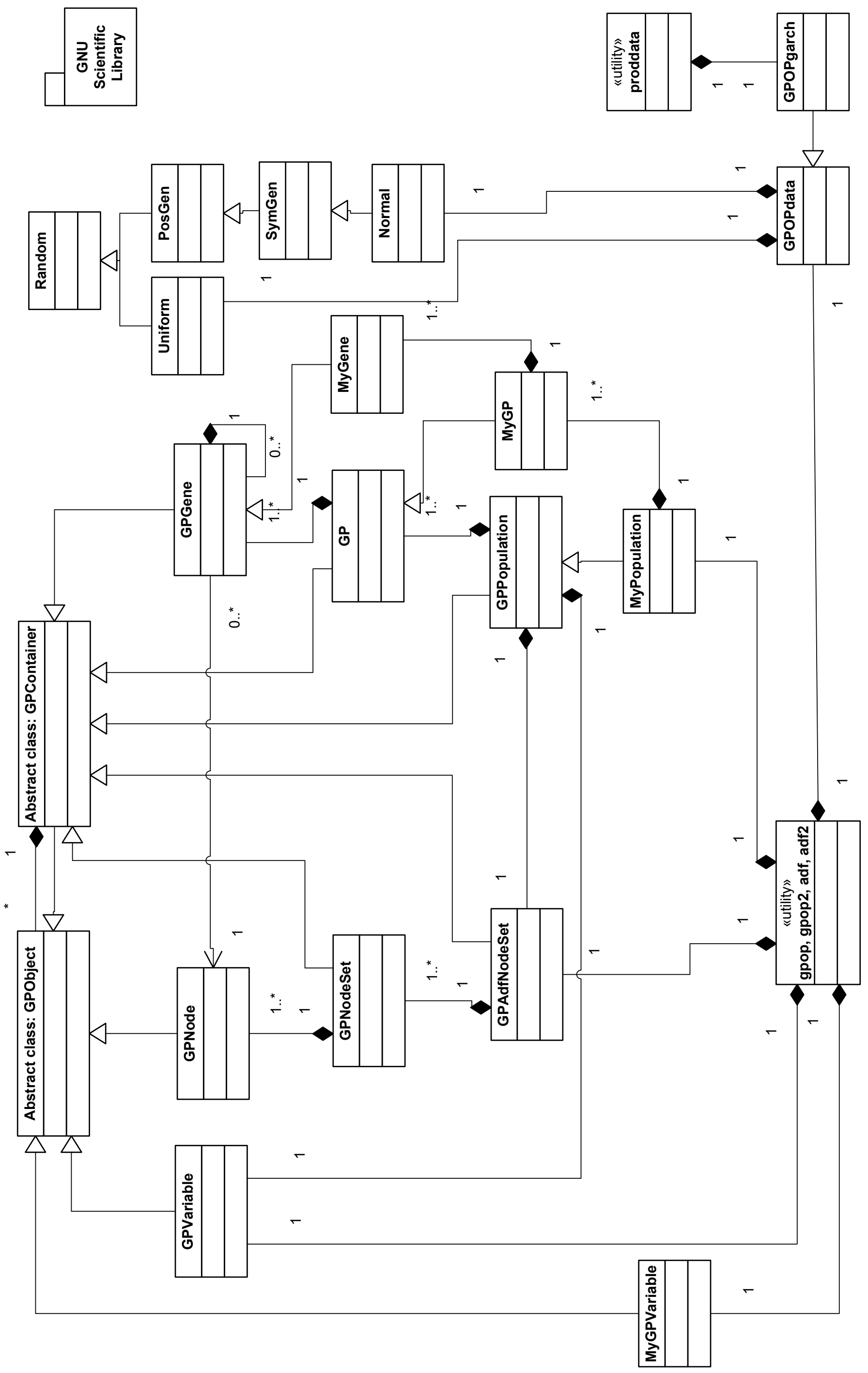

Figure 6.4: UML diagram of the whole program

Chapter 7

Results and statistics

This chapter shows the results of the best genetic programs found. In the early stages I have tried to figure out which is the most promising configuration. Therefore the chapter starts with a statistical test of different configurations. Afterwards the best genetic program found is introduced and some properties of this equation are shown. At the end, my result is compared with the three existing approaches described in chapter 4.

7.1 Statistics of the identification of the best configuration

7.1.1 Setting of the test environment

Each of the two configurations from 5.2.2 (with and without mutation) is used with the three bloat control methods described in chapter 5.2.2. Additionally, all 6 resulting approaches are tried with a Student t distribution and with a standard normal distribution. This leads to 12 different test cases where each test is repeated 25 times to get statistically relevant results. Table 7.1 shows all different test cases.

Due to the huge number of test cases ($12 * 25 = 300$), the control parameters (compare with chapter 3.2.6) have been chosen as run-time decreasing and as little memory consuming as possible:

1. The configuration without mutation is used with a population size of 30000 and for 50 generations.

2. The configuration with mutation is used with a population size of 3000 and for 500 generations.

3. In the depth limiting bloat control method, the *maximum depth* parameter is set to 7.

No.	configuration	bloat control	cdf used
1	without mutation	depth limiting	normal
2	without mutation	depth limiting	student-t
3	without mutation	parsimony pressure	normal
4	without mutation	parsimony pressure	student-t
5	without mutation	death by size	normal
6	without mutation	death by size	student-t
7	with mutation	depth limiting	normal
8	with mutation	depth limiting	student-t
9	with mutation	parsimony pressure	normal
10	with mutation	parsimony pressure	student-t
11	with mutation	death by size	normal
12	with mutation	death by size	student-t

Table 7.1: Test cases for configuration identification

4. In the death by size and the parsimony pressure bloat control method, the maximum depth is set to 20.

5. The *penalty* parameter in the parsimony pressure bloat control method is set to only 20000.

7.1.2 Test results

Table 7.2 shows the mean and the standard deviation of the results as well as the average execution time. The computer I have used is an AMD Athlon XP 2200+ CPU with 512 MByte DDRAM. To draw statistically correct conclusions, the results are compared according to Student's t-test. This test shows whether two samples come from the same population. This statistical test works under the assumption of a normal distribution of the universe, that both samples are statistically independent and that the population of both have identical variances. According to [Vie97] it applies the following formula:

$$Z = \frac{\overline{X}_m - \overline{Y}_n - \mu_x + \mu_y}{\sqrt{(\frac{1}{m} + \frac{1}{n})\frac{(m-1)S_m^2 + (n-1)S_n^2}{m+n-2}}} \sim t_{m+n-2} \tag{7.1}$$

where $X \sim N(\mu_x, \sigma^2)$ and $Y \sim N(\mu_y, \sigma^2)$ are the two stochastic variables, S_y^2 and S_x^2 are the variances of the sample, and $\overline{Y}_n$ and $\overline{Y}_m$ are the averages of the sample.

The hypothesis $H_0 : \mu_x = \mu_y$ has to be abolished, when the following equation holds:

$$Z = \left| \frac{\overline{x}_m - \overline{y}_n}{\sqrt{(\frac{1}{m} + \frac{1}{n})\frac{(m-1)s_m^2 + (n-1)s_n^2}{m+n-2}}} \right| \geq t_{m+n-2;1-\frac{\alpha}{2}} \tag{7.2}$$

with the probability α of rejecting a correct null hypothesis.

No.	Mean	Deviation	Average time
1	0.01532	0.00126	158
2	0.01545	0.00144	190
3	0.01875	0.00183	126
4	0.01825	0.00137	127
5	0.04356	0.00455	12
6	0.04361	0.00342	15
7	0.01853	0.00294	204
8	0.02057	0.00525	205
9	0.02512	0.00917	109
10	0.02978	0.01024	74
11	0.06559	0.00925	3
12	0.06884	0.01006	3

Table 7.2: Results of configuration identification

No	1	2	3	4	5	6	7	8	9	10	11	12
1	-	74.1	1.0	7.3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	74.1	-	2.6	16.4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3	1.0	2.6	-	27.2	0.0	0.0	0.9	0.1	0.2	0.0	0.0	0.0
4	7.3	16.4	27.2	-	0.0	0.0	0.0	0.0	0.1	0.0	0.0	0.0
5	0.0	0.0	0.0	0.0	-	96.5	0.0	0.0	0.0	0.0	0.0	0.0
6	0.0	0.0	0.0	0.0	96.5	-	0.0	0.0	0.0	0.0	0.0	0.0
7	0.0	0.0	0.9	0.0	0.0	0.0	-	9.7	3.2	0.0	0.0	0.0
8	0.0	0.0	0.1	0.0	0.0	0.0	9.7	-	28.1	0.3	0.0	0.0
9	0.0	0.0	0.2	0.1	0.0	0.0	3.2	28.1	-	8.5	0.0	0.0
10	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	8.5	-	0.0	0.0
11	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	-	24.1
12	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	24.1	-

Table 7.3: Results of the t-statistic in percent

The result of the t-test of each tuple of configuration is shown in table 7.3. This table shows the probability in percent of α - the probability of rejecting a correct null hypothesis. If α is higher than 5%, I have assumed that the two configurations are of the same quality. Additionally this table shows which configuration is better with respect to the mean value according to table 7.2. A grey entry signifies that the row configuration is better (on average) than the column configuration. If the entry is black, it is vice versa.

The configuration with mutation leads in all cases to a significantly worse result than the configuration without mutation. Also the execution time is most of the time longer with mutation. Therefore, I have decided to prefer the configuration without mutation for all further experiments.

The death by size bloat control method doesn't work well in my analysis. It kills all the long genetic programs immediately and at the same time the entire gene pool. At the end a couple of very small genetic programs survive with a very bad performance. The execution time is extremely short. The depth limiting bloat control method performs in most cases equally well as the parsimony pressure method, but it takes on average twice as much time. Therefore the Penalty factor has been set to low to allow a suitable judgement. I will use both methods in my further experiments. Death by size will be discarded, due to its poor performance.

The usage of the Student-t or the normal cumulative distribution function as function in the Genetic Programming algorithm doesn't lead to significantly different results in all cases. This might be because the normal distribution is a special case of the Student-t distribution. The additional properties of the Student-t distribution do not lead to better results. Usually the execution time is slightly higher with the Student-t distribution.

7.1.3 Utilization of the ADFs and the hybrid approaches

In order to identify which approach (compare with chapter 5.2.4) is the best one, all approaches have been tested with the best configurations. The standard with ADF approach is even tried with all configurations. In the case of significantly different results, this would show that there are some interdependencies.

Table 7.4 shows the performance of ten runs for each configuration with the ADF approach. It is worse than the average of the standard configuration in all cases. I conclude therefore that the approach with ADF does not lead to better results. This might be because it is not possible to divide the problem into easily solvable subproblems. Another conclusion is that the different configurations lead to comparable results as before. I assume that this is a general pattern.

The hybrid approach was not successful. In both cases with and without ADFs it has not worked in the way that I expected. In all cases results are worse than with the simpler methods. Another drawback was the usage of the Paretian stable distribution function. It is complicated to calculate (with numerical integrals), which takes a lot of time (20 times longer than with a normal distribution function), but has not led to

No.	Mean	Deviation	Average time
1	0.02183885	0.00383879	131
2	0.02224098	0.00477594	144
3	0.02579130	0.00565985	131
4	0.02666881	0.00390301	142
5	0.04172123	0.00467133	63
6	0.04438526	0.00412538	87
7	0.02979943	0.01253381	288
8	0.03159794	0.01802966	258
9	0.04453466	0.01503629	56
10	0.03586901	0.00740740	60
11	0.06872159	0.00799661	9
12	0.06238697	0.01195662	11

Table 7.4: Fitness values of the ADF approach

better results (not even results of equal quality).

7.2 Comparison of the best equation found with the original process

7.2.1 Setting of the Genetic Algorithm

To get the best possible results, the most promising configurations have been tested with all approaches. Additionally, the control parameters are set to the maximum values, where preliminary tests have indicated that they will lead to best results.

1. This leads to a maximum population size of 50000,
2. the number of generations is still set to 50,
3. the maximum depth in the depth limiting bloat control method is set to 10
4. and the penalty factor in the parsimony pressure bloat control method is set to 200000.

This configuration does not overflow my memory with 512 MByte RAM and only takes reasonable time (around 10 hours) on the average.

7.2.2 The result

The best equation has been found after months of exhaustive testing with the most promising configurations. It has been discovered with the depth limiting configuration without mutation and the simple approach without ADFs. The best found equation is (with some simplifications):

$$\begin{gathered}
f_1 = \max((\max(\mathrm{NORMAL}(\min(\ln(\tfrac{S}{X}); dt \cdot r)); \tfrac{S}{X}) - (\max(\min(a_1 \cdot b_1; a_1^{dt \cdot \sigma^2)} + \\
a_1^{r+\frac{h_t}{\sigma}} \cdot (dt \cdot \sigma^2 + dt \cdot \sigma^2); \max(\ln(\tfrac{S}{X}) + \\
\min(dt \cdot \sigma^2; pi + r); dt \cdot \sigma^2 \cdot (dt \cdot \sigma^2 + dt \cdot \sigma^2))))) \cdot (\min(dt \cdot \sigma^2; \\
(dt + dt \cdot \sigma^2) \cdot \tfrac{S}{X} \cdot \sigma^2 \cdot \tfrac{S}{X}) + \max(\ln(\tfrac{S}{X}); \min(\min(b_1; dt \cdot \sigma^2); dt \cdot r))) + \\
\min(\ln(\tfrac{S}{X}); \min(\ln(\tfrac{S}{X}); \min(\min((dt + b_1 + e) \cdot \max(\tfrac{S}{X}; 1) \cdot \sigma^2; dt \cdot r); \\
\mathrm{NORMAL}(1 - (\tfrac{S}{X}))))); \max((\max(\max(\ln(\tfrac{S}{X}); \min(dt + \max(\tfrac{S}{X}; 1); dt \cdot r)); \tfrac{S}{X}) - \\
(\max(a_1^{r+\frac{h_t}{\sigma}} \cdot (dt \cdot \sigma^2 + dt \cdot \sigma^2) + a_1^{r+\frac{h_t}{\sigma}} a_1 b_1; \max(dt \cdot \sigma^2; a_1^{r+\frac{h_t}{\sigma}} (dt \cdot \sigma^2 + dt \cdot \sigma^2))))) \\
(\min(((\ln(\tfrac{S}{X})) \cdot \tfrac{S}{X} + dt) \cdot \sigma^2; (dt + \ln(\tfrac{S}{X})) \cdot \tfrac{S}{X} \cdot \sigma^2 \cdot \tfrac{S}{X}) + \frac{\max((ln(\frac{S}{X}))}{(r+\frac{h_t}{\sigma})} \cdot \sigma^2; \min(a_1^{r+\frac{h_t}{\sigma}} \cdot \\
(dt \cdot \sigma^2 + dt \cdot \sigma^2); dt \cdot r))); \tfrac{S}{X} - ((dt \cdot \sigma^2 + dt \cdot \sigma^2)^{dt \cdot \sigma^2)} \\
-(\max(\frac{\min(\ln(\frac{S}{X}); Lambda + \frac{S}{X}) + a_1^{r+\frac{h_t}{\sigma}} \cdot (dt \cdot \sigma^2 + dt \cdot \sigma^2)}{b_1 + e}; \min(dt \cdot \sigma^2; \ln(\tfrac{S}{X}))))))
\end{gathered}$$

The complete equation in Microsoft EXCEL style:

```
max((max(STANDNORMVERT(min(min(ln(S/X);ln(S/X));dt*r));S/X)-
(max(min(a1*b1;a1^(dt*Sigma2))+a1^(r+ht/Sigma)*(dt*Sigma2+dt*Sigma2);
max(min(ln(S/X);ln(S/X))+min(dt*Sigma2;pi+r);dt*Sigma2*
(dt*Sigma2+dt*Sigma2)))))*(min(dt*Sigma2;((dt+dt*Sigma2)*
(S/X-0)*Sigma2)/(1/S/X))+max(ln(S/X);min(min(b1;dt*Sigma2);dt*r)))+
min(ln(S/X);min(ln(S/X);min(min((dt+b1+e)*max(S/X;1)*Sigma2;dt*r);
STANDNORMVERT(1-(max(S/X-0;S/X))))));max((max(max(ln(S/X);
min(dt+max(S/X;1);dt*r));S/X)-(max(a1^(r+ht/Sigma)*
(dt*Sigma2+dt*Sigma2)+a1^(r+ht/Sigma)*a1*b1;max(dt*Sigma2;
a1^(r+ht/Sigma)*(dt*Sigma2+dt*Sigma2)))))*(min(((ln(S/X))/(1/S/X)+dt)*
Sigma2;((dt+ln(S/X))*(S/X-0)*Sigma2)/(1/S/X))+max((ln(S/X))/
(r+ht/Sigma)*Sigma2;min(a1^(r+ht/Sigma)*(dt*Sigma2+dt*Sigma2);dt*r)));
max(S/X-0;S/X)-((dt*Sigma2+dt*Sigma2)^(dt*Sigma2))-(max((min(ln(S/X);
Lambda+S/X)+a1^(r+ht/Sigma)*(dt*Sigma2+dt*Sigma2))/(b1+e);
min(dt*Sigma2;ln(S/X))))))
```

The quality of the result in relation to different numbers of data points is listed in table 7.5. With more data points, the RMSE even gets better. This is a sign that no over-fitting has occurred. Under the assumption that the underlying follows in reality a GARCH process and the prices of the options are calculated (wrongly) according to the Black-Scholes model, table 7.5 also shows the $RMSE$ of the Black-Scholes formula. The newly found equation is therefore about 25% better suited to find the price of a European option.

Number of data points	RMSE of best equation	RMSE of Black-Scholes	Improvement
1000	0.010656375	0.014230704	25.12 %
10000	0.010115501	0.013133606	22.98 %

Table 7.5: Results of the best found equation

Name	dt in days	S/X
data-095-30	30	0.95
data-100-30	30	1.00
data-105-30	30	1.05
data-095-60	60	0.95
data-100-60	60	1.00
data-105-60	60	1.05
data-095-90	90	0.95
data-100-90	90	1.00
data-105-90	90	1.05

Table 7.6: Simulation of the process

7.2.3 The original process

To get some idea about the original GARCH process, I have simulated the option prices 100 times with 9 different settings. The values for S/X and dt varies according to table 7.6 and the values for r, σ^2, h_t/σ, $a1$, $b1$ and λ have been set to 0.05, 0.5, 1.0, 0.8, 0.15 and 0.001, respectively.

In table 7.7, the standard deviation of the original process, the RMSE of the best equation and the RMSE of the Black-Scholes formula are shown with respect to the simulated data described above. The average standard deviation is 0.00085 and the RMSE is an order of magnitude higher. This seems to be realistic, because it is not possible to construct a formula which reaches a value that is lower than the standard deviation of a constant data set. In this case, the RMSE of the Black-Scholes formula is three times inferior on average. Depending on the value of S/X and dt, the RMSE varies between 0.00290 and 0.02019. This indicates that better results are still possible.

Figure 7.1 shows that the RMSE of the best equation found behaves regularly in comparison to the RMSE of the Black-Scholes formula (compare with figure 7.2). The best found equation has therefore no significant bad properties at a specific point.

A different error measure which may be of economical interest is the Mean Absolute Percentage Error (MAPE) which is defined as follows:

$$MAPE = \frac{1}{N}\sum_{n=1}^{N}|\frac{e(n)}{y(n)}| \tag{7.3}$$

Name	Std. deviation	RMSE	RMSE Black-Scholes
data-095-30	0.00079	0.00995	0.02955
data-095-60	0.00100	0.00919	0.04411
data-095-90	0.00094	0.01070	0.05394
data-100-30	0.00074	0.01842	0.03168
data-100-60	0.00097	0.00290	0.04587
data-100-90	0.00090	0.00298	0.05554
data-105-30	0.00068	0.02019	0.03032
data-105-60	0.00091	0.00885	0.04524
data-105-90	0.00070	0.01456	0.00460
Average	0.00085	0.01086	0.03787

Table 7.7: Standard deviation and RMSE

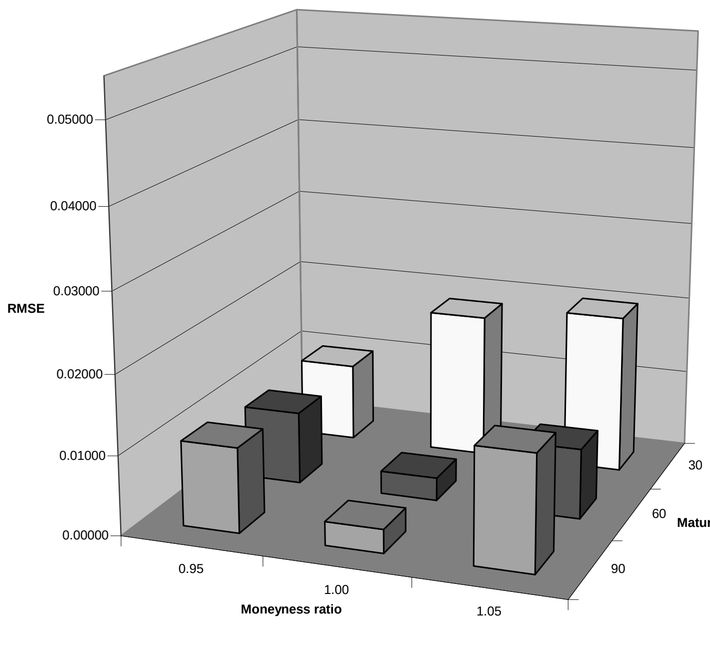

Figure 7.1: RMSE - best equation found

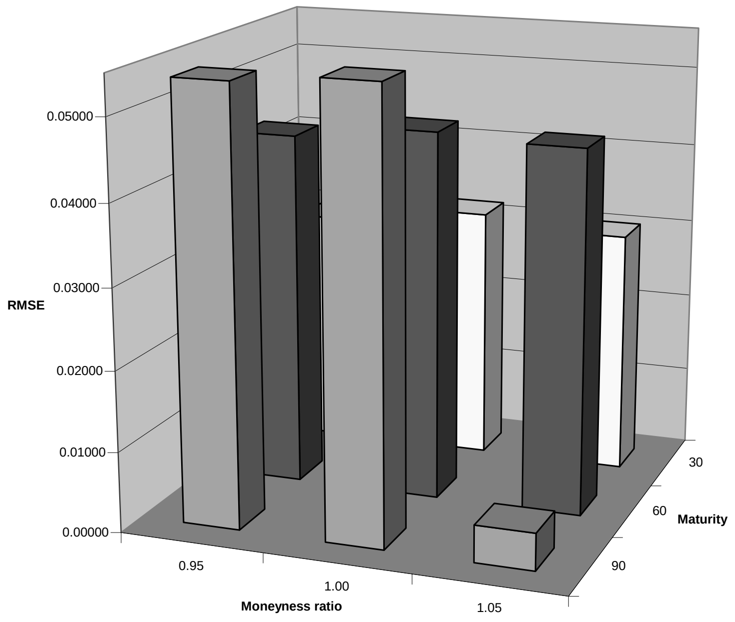

Figure 7.2: RMSE - Black-Scholes

where N is the number of data points, e is the error term and y is the Genetic Programming result.

Table 7.8 shows the different option prices the three methods get with the 9 constant data sets. In all cases the newly found equation leads to better results than the Black-Scholes formula. Still the MAPE is between 2.43% (which seems to be acceptable) and 27.66% (which seems to be too high).

In figure 7.3 one can also observe the much higher relative differences at an expiration time of 30 days than at 90 days in all cases. The RMSE (compare with figure 7.1 is much better behaved than the MAPE, which is logical, because the driving force (the fitness function) is in my case the RMSE. To get better relative differences I should have used the MAPE as fitness function or maybe a combination of both. But this would increase the calculation complexity. With the Black-Scholes formula such a relation is not observable. Figure 7.4 shows that the Black-Scholes formula seems to have a significantly higher MAPE around the at the money point (where S/X equals 1.0).

7.3 Comparison of the results with other approaches

This section compares the equation generated via Genetic Programming with the three existing approaches described in chapter 4.

Name	Average Result GARCH	Result Equation	MAPE	Result Black-Scholes	MAPE
data-095-30	0.04292	0.05284	23.12 %	0.07246	12.78 %
data-095-60	0.06824	0.07737	13.38 %	0.11234	32.96 %
data-095-90	0.08942	0.10008	11.92 %	0.14335	45.24 %
data-100-30	0.06655	0.08496	27.66 %	0.09822	11.45 %
data-100-60	0.09404	0.09131	2.91 %	0.13990	157.67 %
data-100-90	0.11655	0.11939	2.43 %	0.17209	228.22 %
data-105-30	0.09777	0.11795	20.64 %	0.12808	14.69 %
data-105-60	0.12511	0.13391	7.04 %	0.17034	64.29 %
data-105-90	0.19860	0.18406	7.32 %	0.20314	6.21 %
Average	0.09991	0.10687	12.94 %	0.13777	29.27 %

Table 7.8: Average results of the process

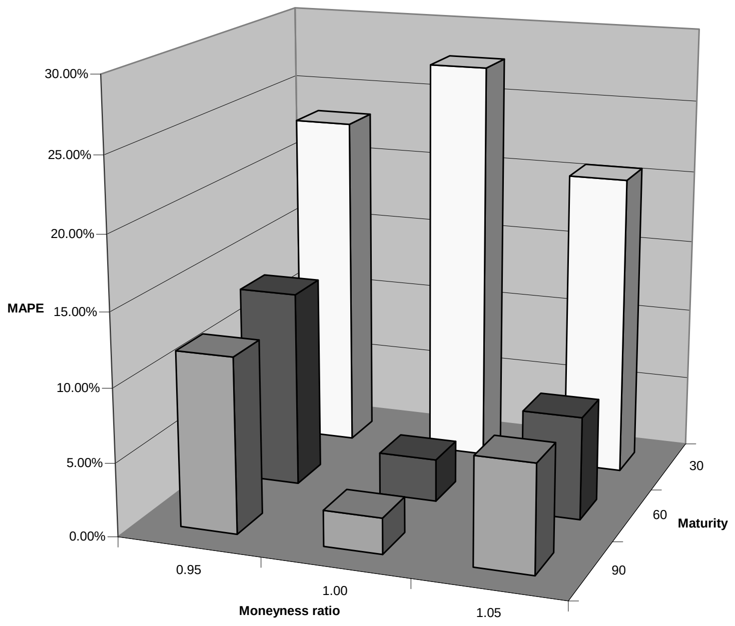

Figure 7.3: MAPE - best equation found

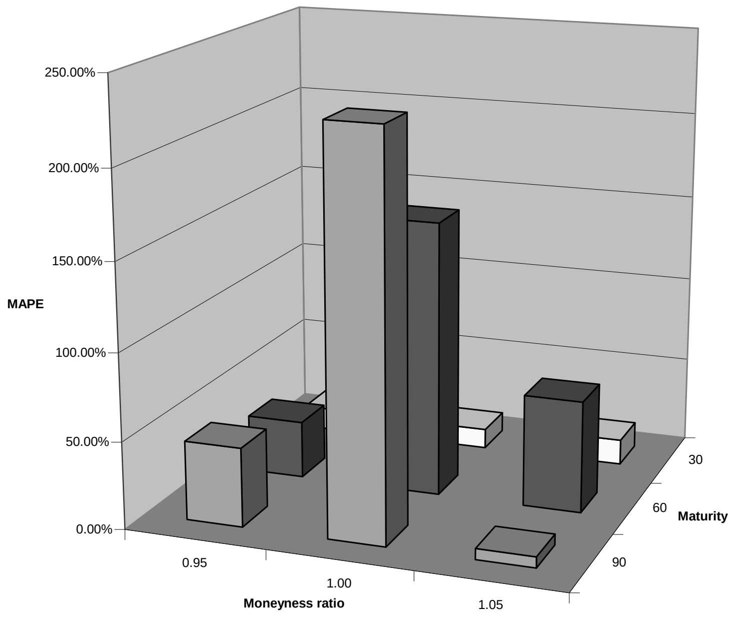

Figure 7.4: MAPE - Black-Scholes

7.3.1 Comparison of the result with [Han98]

In [Han98] the best found neural network has a $RMSE$ of $5.63 * 10^{-4}$ which is much better than the $101.16 * 10^{-4}$ I get in my result equation. The analysis of the simulated price paths (compare with table 7.7) shows that even the standard deviation is higher with a value of $8.5 * 10^{-4}$. Therefore I conclude that a result with this accuracy is not possible in my setting.

Another difference is that I have simulated the data in a much wider range. In [Han98] the expiration time has been simulated between 1 and 30 days and the annualized unconditional variance between 0.01 and 0.16. In comparison, I have used the range $[1, 90]$ for the maturity and $[0.01, 1.00]$ for the unconditional annualized variance, respectively. These data simulate a higher number of options and offer a broader application, but lead at the same time to a higher variance of the original Monte Carlo simulation.

7.3.2 Comparison of the result with [DS01]

[DS01] price options where the underlying follows a GARCH process via a Markov chain approximation. They use fixed values for all variables except for the expiration time and the moneyness ratio. This leads to a lower variance of the original process than in my analysis. What is more complex in [DS01] is the usage of the NGARCH(1,1) model, where the leverage effect is considered too. Still the results can be compared.

At different maturities they get an error between $6 * 10^{-3}$ and $12 * 10^{-3}$ which is as good than my average value of 10.116^{-3}. It seems that the approach of [DS01] and my new approach are equally suitable to price options in a GARCH framework. I think that my result of using only a single formula is easier than solving a 3 dimensional Markov chain. Therefore it is more likely to be used in practise.

7.3.3 Comparison of the result with [Keb99]

The quality of the results is not comparable because Keber uses a different type of option with a different model for the underlying. In my case it is not possible to analyze the best equation found theoretically, because it is just an equation which fits best according to the fitness function. I think in general one should not expect, to get a short equation via the Genetic Programming approach which lends itself well to economic interpretation. Maybe [Keb99] was only lucky, or he restricted the problem domain so that no other type of equation was able to emerge.

Still this thesis has shown that Genetic Programming is also suitable for a complex type of model (GARCH) and with a broad range of input parameters. [Keb99] has shown that this approach is also suitable for American options.

Chapter 8

Conclusion

8.1 New approaches

This master thesis uses for the first time the Genetic Programming approach to price options which follow a GARCH process. The data sample is produced via the well known Monte Carlo simulation method. The range of the simulated data is much higher than in other existing approaches. This leads to the situation that all European call options with an expiration time of up to 90 trading days are covered. The only additional assumption is that the underlying follows a GARCH(1,1) process, where one can find a lot of empirical evidence that this assumption is realistic.

Also new is the usage of different settings for the Genetic Programming approach in the area of option pricing. A configuration with and without mutation is tested with different bloat control methods. Additionally three different distribution functions are used. As a state-of-the-art feature, the new approach of Automatic Defined Functions is utilized. A hybrid model with a Black-Scholes analog function is tried, too. It was expected that this hybrid model would harmonize well with the Automatic Defined Functions.

8.2 Summary of the result

The experiments lead to the conclusion that the easier settings are more successful than the complicated ones. Mutation does not work well and the simplest bloat control methods (depth limiting and parsimony pressure) are successful. At the same time the use of a normal distribution function seems sufficient, because the use of others does not lead to better results. Even the use of Automatic Defined Functions and the hybrid approach, respectively does not improve the results.

The best equation found is about 25% better (in term of RMSE) suited to price European call options following a GARCH process than the Black-Scholes formula. Over-fitting was avoided and the behavior of the equation is regular. It does not have any specific area where the quality of the equation is low. It offers therefore reasonable prices for all European call options which are traded at exchanges. The conclusion is therefore that Genetic Programming is well suited to find pricing equations in this environment.

8.3 Future issues

The Genetic Programming approach has the property that with higher computer power, the results tend to get better. Therefore, in a few years a repetition of my tests with a more powerful computer might lead to better results. The field of Genetic Programming is still an active area of research. New results may show new configuration settings which are especially beneficial in the environment of option pricing. Additionally, there are still many more configurations possible with the Automatic Defined Function approach which I could not test in this master thesis due to limited time. Especially the evolutionary finding of the best architecture seems to be promising, but resource expensive at the same time.

The field of option pricing is also an area of intensive research. New models emerge, like the NGARCH and the EGARCH to name just a few. These models also capture the leverage effect (compare with [DS01]). The more complicated the models will become, the less likely it is that a closed formula can be derived. I conclude therefore that in the future there will be increasing necessity for guided empirical model finding methods like Genetic Programming.

List of Figures

List of Tables

Bibliography

[BBG97] Phelim Boyle, Mark Broadie, and Paul Glasserman. Monte carlo methods for security pricing. *Journal of Economic Dynamics and Control*, pages 1267–1321, 1997.

[BCK92] Tim Bollerslev, Ray Y. Chou, and Kenneth F. Kroner. ARCH modeling in finance: a review of the theory and empirical evidence. *Journal of Econometrics*, 52:5–95, 1992.

[BFM97] Thomas Bäck, David B. Fogel, and Zbigniew Michalewicz. *Handbook of Evolutionary Computation.* Oxford University Press, New York, 1997.

[BNKF98] Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone. *Genetic Programming - An Introduction.* Morgan Kaufmann, San Francisco, California, USA, 1998.

[Bol86] Tim Bollerslev. Generalized autoregressive conditional heteroskedasticity. *Journal of Econometrics*, 31:307–327, 1986.

[BS73] Fischer Black and Myron Scholes. The pricing of options and corporate liabilities. *Journal of Political Economy*, 81:637–659, 1973.

[Dav02] Robert B. Davis. Newran02b - a random number generator library. Technical report, Statistics Research Associates Limited, Wellington, New Zealand, 2002.

[DS95] Jin-Chuan Duan and Jean-Guy Simonato. Empirical Martingale Simulation for Asset Prices. *CIRANO Working Papers*, 95s-43, 1995.

[DS01] Jin-Chuan Duan and Jean-Guy Simonato. American option pricing under GARCH by a markov chain approximation. *Journal of Economic Dynamics & Control*, 25:1689–1718, 2001.

[Dua95] Jin-Chuan Duan. The GARCH option pricing model. *Mathematical Finance*, 5(1):13–32, 1995.

[Eng82] Robert F. Engle. Autoregressive conditional heteroskedasticity with estimates of the variance of u.k. inflation. *Econometrica*, 50:987–1008, 1982.

[Fra94] Adam P. Fraser. Genetic Programming in C++. Technical Report 040, University of Salford, Cybernetics Research Institute, 1994.

[G$^+$04] M. Galassi et al. *GNU Scientific Library Reference Manual*. Network Theorie Ltd., Bristol, United Kingdom, 2 edition, 2004.

[GS96] Alois L. J. Geyer and Walter S. A. Schwaiger. GARCH Effekte in der Optionsbewertung. *Zeitschrift für Betriebswirtschaft*, 65(5):534–540, 1996.

[Han97] Michael Hanke. Neural Network Approximation of Option Pricing Formulas for Analytically Intractable Option Pricing Models. *Journal of Computational Intelligence in Finance*, 5(5):20–27, Sep 1997.

[Han98] Michael Hanke. *Optionsbewertung mit neuronalen Netzen*. Dissertation, Wirtschaftsuniversität Wien, 1998.

[Hol75] John H. Holland. *Adaption in Natural and Artificial Systems*. University of Michigan Press, 1975.

[Hul02] John C. Hull. *Options, futures, and other derivatives*. Prentice Hall International, Inc., Upper Saddle River, New Jersey, USA, fifth edition, 2002.

[K$^+$03] John R. Koza et al. *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. Kluwer Acadamic Publishers, 2003.

[KBAK99] John R. Koza, Forrest H. Bennett III, David Andre, and Martin A. Keane. *Genetic Programming III, Darwinian Invention and Problem Solving*. Morgan Kaufmann Publishers, San Francisco, California, USA, 1999.

[Keb99] Christian Keber. Option Pricing with the Genetic Programming Approach. *Journal of Computational Intelligence in Finance*, 7(6):26–36, 1999.

[Koz92] John R. Koza. *Genetic Programming*. The MIT Press, Cambridge, Massachusetts, USA, 1992.

[Koz94] John R. Koza. *Genetic Programming II, Automatic Discovery of Reuseable Programs*. The MIT Press, Cambridge, Massachusetts, USA, 1994.

[Mic92] Zbigniew Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer, Berlin, Germany, 1992.

[Nef00] Salih N. Neftci. *An Introduction to the Mathematics of Financial Derivatives*. Acadamic Press, San Diego, California, USA, second edition, 2000.

[Nol97] John P. Nolan. Numerical calculation of stable densities and distribution functions. *Commun. Statist. - Stochastic Models*, 13(4):759–774, 1997.

[Obj03] Object Management Group, Inc. *OMG Unified Modeling Language Specification, Ver. 1.5*, March 2003.

[PL04] Liviu Panait and Sean Luke. Alternative Bloat Control Methods. In *Lecture Nodes in Computer Science 3103*, pages 630–641. Springer-Verlag, 2004.

[Rey92] Craig W. Reynolds. An evolved, vision-based behavioral model of coordinated group motion. In Meyer and Wilson, editors, *From Animals to Animats (Proceedings of Simulation of Adaptive Behaviour)*. MIT Press, 1992.

[Sin94] Andy Singleton. Genetic Programming with C++. *BYTE Magazin*, February 1994.

[Str98] Bjarne Stroustrup. *The C++ Programming Language.* Addison Wesley Longman, Reading Mass, USA, third edition, 1998.

[Vie97] Reinhard K. Viertl. *Einführung in die Stochastick*. Springer-Verlag, Wien, second edition, 1997.

[Wei97] Thomas Weinbrenner. The Genetic Programming Kernel. Technical report, Institute for Mechatronics, Technical University of Darmstadt, 1997.

[Zol86] V. M. Zolotarev. One-dimensional Stable Distributions. *Amer. Math. Soc. Transl. of Math. Monographs*, 65, 1986.