Financial Application of Genetic Programming

by

Magnus Johansson

LIU-IDA/LITH-EX-A--09/010--SE

2009-02-26
Final Thesis

Financial Application of Genetic Programming

By

Magnus Johansson

2009-02-26

LIU-IDA/LITH-EX-A--09/010—SE

Examiner: Kristian Sandahl
PREFACE

After four years of studies at the university it was interesting to get an opportunity to use some of this theoretic knowledge in a more practical manner. This final thesis has been done on the company QuantSystem which is located in Växjö. I would like to thank QuantSystem for giving me the opportunity to do my final thesis there. The work has been interesting and given me knowledge which I will have good use of in my future working life. I hope the personnel at QuantSystem can have use of my work in the future as well.

Linköping, 22 January 2009.
SUMMARY
With the increasing speeds of modern processors the possibility of using genetic programming for problems with a huge amount of data has become feasible. One area where people over the course of time have been interested in looking for pattern is in the financial markets. Due to the nature of financial markets it is very hard to find patterns with traditional techniques. It is hoped that genetic programming can find these patterns that can’t be found in other ways, if they exist. This report studies genetic programming and a system called TSL that creates trading models with the help of genetic programming. TSL is built on a genetic programming software called Discipulus which is a very fast machine code based regression and classification tool.

The first step before a run can take place is to collect the financial data that the models will be built from. During this work the data has been taken from TradeStation which is a system used for analyzing and trading the financial markets. After this is done TSL must be set up for the run. It has a lot of different parameters for the user to configure. When the run is over some of the models are saved and these can be tested in TradeStation to see their performance on another time period. If it gives a satisfactory result the models can be used for live trading.

During the work I have focused on two futures contracts, Standard & Poor’s 500 E-mini contract and the British Pound contract. On these instruments extensive testing has been made but I have not been able to find any models that return risk adjusted excess returns during my work. There is a possibility thou that such systems actually has been produced during the evolution but due to flaws in the saving mechanism in TSL some of the most promising looking models have not been saved.
## Table of Contents

1. **INTRODUCTION** ........................................................................................................................................ 1
   
   1.1  BACKGROUND ........................................................................................................................................ 1
   
   1.2  OBJECTIVE ............................................................................................................................................ 1
   
   1.3  TARGET GROUP .................................................................................................................................... 1
   
   1.4  METHOD ................................................................................................................................................ 1
   
   1.5  RESTRICTIONS .................................................................................................................................... 1
   
   1.6  REPORT STRUCTURE ............................................................................................................................ 2

2. **GENETIC PROGRAMMING** ....................................................................................................................... 3
   
   2.1  WHAT IS GENETIC PROGRAMMING? ....................................................................................................... 3
       
       2.1.1  How does GP work ........................................................................................................................ 5
       
       2.1.2  History ............................................................................................................................................. 7
       
       2.1.3  Theory .............................................................................................................................................. 10
   
   2.2  BASIC CONCEPTS .................................................................................................................................. 13
       
       2.2.1  Population ....................................................................................................................................... 13
       
       2.2.2  Terminal set ..................................................................................................................................... 13
       
       2.2.3  Function set ..................................................................................................................................... 14
       
       2.2.4  Fitness function ............................................................................................................................... 15
   
   2.3  GENETIC OPERATORS ........................................................................................................................... 15
       
       2.3.1  Crossover ....................................................................................................................................... 15
       
       2.3.2  Mutation ......................................................................................................................................... 17
       
       2.3.3  Reproduction .................................................................................................................................. 17
   
   2.4  INTRONS ................................................................................................................................................ 18
       
       2.4.1  Why does introns exist in GP ......................................................................................................... 18
   
   2.5  DISCIPULUS ............................................................................................................................................ 19
       
       2.5.1  Fitness calculation ............................................................................................................................ 21
       
       2.5.2  Instructions ...................................................................................................................................... 21
       
       2.5.3  Other settings of interest .................................................................................................................. 22

3. **FINANCIAL APPLICATION** ....................................................................................................................... 24
   
   3.1  AUTOMATED TRADING .......................................................................................................................... 24
   
   3.2  TSL .......................................................................................................................................................... 25
       
       3.2.1  Preprocessing ................................................................................................................................... 25
       
       3.2.2  Settings .......................................................................................................................................... 26
       
       3.2.3  Example run .................................................................................................................................... 30
4 MY WORK .................................................................................................................................................. 34
4.1 COURSE OF ACTION .............................................................................................................................. 34
   4.1.1 Working with TSL .......................................................................................................................... 34
   4.1.2 Working with Discipulus ............................................................................................................... 35
   4.1.3 Advanced features in TSL .......................................................................................................... 37
5 RESULTS ..................................................................................................................................................... 38
   5.1 USABILITY .......................................................................................................................................... 38
   5.2 PERFORMANCE ................................................................................................................................... 38
   5.3 MODELS ......................................................................................................................................... 39
6 CONCLUSION ........................................................................................................................................ 42
   6.1 FINAL DISCUSSION ......................................................................................................................... 42
7 REFERENCES ......................................................................................................................................... 43
   7.1 PRINTED SOURCES .......................................................................................................................... 43
   7.2 ELECTRONIC SOURCES ............................................................................................................... 43
1 INTRODUCTION

This chapter presents the background, objective, target group, method, restrictions and an overview of the structure of the report.

1.1 BACKGROUND

As the world and the financial markets have become more computerized during the last decade the concept of automated trading has gotten more attention. The idea is compelling because if good models could be found they can work independently and generate excess gains. If there are patterns in the markets these may be of such a complex nature that they are hard for humans to find with ordinary numerical techniques. Instead genetic programming is used to try to find these patterns. TSL is a software package that generates trading models based upon genetic programming.

1.2 OBJECTIVE

This reports objective is to examine and analyze the software package TSL and the quality of the models that it creates. To be able to do this good knowledge about genetic programming is necessary which is why a big part of the report is about genetic programming.

1.3 TARGET GROUP

This report can be read by anyone that has an interest in automated trading or genetic programming. To be able to understand parts of the report a certain level of technical and mathematical knowledge is required but these parts can be skipped if wanted. Basic knowledge about the how the financial markets work is also recommended.

1.4 METHOD

The most important parts of this work have been to gain knowledge about genetic programming and to test TSL. I have gained knowledge about genetic programming from literature and through contact with the creators of Discipulus, which is genetic programming software for regression and classification problems that TSL is built upon. The knowledge about TSL comes from testing, reading the manual and from contact with the creator of the system. The work was divided in 4 parts.

- Gaining knowledge
- Test TSL thoroughly
- Test Discipulus
- Writing the report

1.5 RESTRICTIONS

The testing is restricted by the time I had to do the actual testing. I have only done my testing on two instruments primarily and it would have been preferred if the testing could have been done on more instruments. The work is also restricted by copyright concerns; some parts cannot be shown without violating the copyright of the programs.
1.6 REPORT STRUCTURE
The report is divided into three main parts

- Theory
- Analysis
- Results & Conclusions

The report begins with a theory part which deals with the theory behind genetic programming. The analysis part describes Discipulus and TSL and how my work has been done. In the final part I present the results and the conclusions I have drawn from these.
2 GENETIC PROGRAMMING
This chapter starts with an opening discussion about what genetic programming is and how the principles of evolution work in nature and how genetic programming tries to mimic this process. It also includes the theory behind evolutionary programming and concepts important in this area. The chapter is finished with a part about Discipulus which is a genetic-programming software used for regression and classification problems.

2.1 WHAT IS GENETIC PROGRAMMING?
Genetic programming, which will be called GP throughout this paper, is a technique which is inspired by the theory of evolution. A good starting point for a paper about GP is a brief discussion on evolution in nature to get an overview on what GP is trying to imitate. It is important to note that GP is by no mean an exact copy of the evolution in nature rather it is inspired by it and takes some off the most interesting features of natural evolution and implements into the world of machine learning. A citation\(^1\) from the founder of theory of evolution, Charles Darwin, will start the discussion.

“...if variations useful to any organic being do occur, assuredly individuals thus characterized will have the best chance of being preserved in the struggle for life; and from the strong principle of inheritance they will tend to produce offspring similarly characterized. This principle of preservation, I have called, for the sake of brevity, Natural Selection.”

Charles Darwin, 1859

The citation above states that there are four essential preconditions for the occurrence of evolution by natural selection:

1. Reproduction of individuals in the population
2. Variation that affects the likelihood of survival of individuals
3. Heredity in reproduction
4. Finite resources causing competition

In a simplified model of the genetic code for organisms the DNA can be regarded as a complex set of instructions for creating an organism. A gene is a location in the DNA which can decide an attribute such as what color your hair will be. Variations in this location are called alleles and decide what color your hair actually will be. There is only a small part of the DNA that is engaged in transcriptional activity\(^2\). Those parts are separated by long sequences of DNA for which no function has been identified. These parts are called “junk DNA” or introns and make out the majority of the DNA. Intron in GP is an important topic and will be described in chapter [2.4]. Even thou no function has been identified for introns it does not mean that introns are useless but may have some meaning that we don’t understand yet. Chromosomes are a single piece of DNA and are used in much of the

---

\(^1\) Bahnzaf, Nordin, Keller, Francone (1998)
\(^2\) Transcription is the process of transcribing DNA nucleotide sequence information into RNA sequence information.
literature about evolutionary algorithms as a synonym to an individual in the population. This might be a bit misleading because each cell in every organism of a given species carries a certain number of chromosomes. However one can think of it as one-chromosome individuals.

It is important to distinguish between the appearance of an organism and its genetic constitution. The appearance is described by its *phenotype* and its genetic constitution by its *genotype*. Evolution interacts differently with these two. The genotype of an organism is the DNA of that organism and this is passed from its parents. In other words heredity is passed through the genotype. Also variance goes through the genotype because when the DNA is passed from the parents some part of it may be mutated. Whether it is a beneficial mutation or a possible lethal one for the offspring is another matter. The phenotype on the other hand is the set of observable properties of the individual like its body and its behavior. Natural selection acts on the phenotype and not on genotype so an individual must survive to reproduce.

Exchange of genetic material happens through recombination. DNA from both parents is recombined to produce a new DNA molecule for the offspring. When recombination occurs in nature most of them are of so called homologous nature. This means that when two parents DNA are recombined in the creation of their offspring there are strict rules on how this can happen. For starters exchange can only occur between two identical or almost identical segments of DNA. Secondly homologous crossover can only occur if the two DNA segments to be exchanged can be matched up so that the swap point is at functionally identically points on each strand.

Figure 1 shows a homologous recombination between two individuals where the first gene in the DNA-segment produces a certain protein A, the second gene protein B and the third protein C. The different variations of the genes are the alleles. In this thought-up example both the “owners” of the resulting DNA structures would likely survive. On the other hand with non-homologous recombination the exchange can occur at a random swap point or between non-identical segments of DNA. The latter is the case when individuals of different species mate because they have different DNA structure. The effect of this is that the resulting offspring will most likely die.
GP systems try to imitate the natural evolution by using recombination, which is called crossover in GP, and mutation on the individual.

2.1.1 How does GP work
In GP the individuals are programs and most of the resources in GP systems often go to calculating the different individual’s fitness. This is usually done by running them on different inputs and comparing them to a desired output. This input is called training set and it is on this data the created programs are based. Often a test is done how well the programs work on data that was not part of the creation. This data is called out of sample (OOS) data and if the generated program gives good result both in sample and out of sample it is probably a good system. On the other hand if the results on the training data are good but the result OOS is poor this is likely a problem which is called over fitting. This is a common problem in GP and basically means that the program has curve fitted the training data. There are many techniques for reducing the risk of over fitting and some of these will be discussed in chapter [2.5].

In computerized evolution, like in nature, there is a population of individuals. The fortunate or the fittest survive and can produce offspring and their genetic material is passed on. For a GP search to be able to find better individuals (more fit) there must be some kind of evaluation of the individuals to see which ones are more promising than the others. This is exactly what happens and in GP this evaluation metric is called a fitness function. Individuals with higher fitness have a greater chance of surviving and passing their DNA (their code) to the next generation. In GP there are two common algorithms for how to deal with generations, the *generational* and the *steady state* algorithm. In the generational algorithm there are distinct generations where a new generation completely replaces the old one. This can be compared to species in the nature where only their eggs survive the winter and after each winter a new generation is present. In steady state there is continuous creation of children which replaces older individuals. This has similarities with humans where kids are
born and people are dying continuously. The following steps describe the generational algorithm:

1. Initialize the population.
2. Evaluate each individual in the population and assign a fitness value to each one.
3. Select an individual or individuals in the population with some selection algorithm where the probability that an individual is chosen is proportional to its fitness.
4. Perform genetic operations on the selected individual or individuals.
5. Insert the result to the new generation.
6. Repeat step 3-6 until the new generation is full, then replace the old generation with the new one.
7. If some termination criteria are fulfilled then abort the run and present the best individual otherwise start over from step 2.

In the steady state algorithm a concept called tournaments is used. Tournaments are small scale competitions where the winners of the tournament replace the losers. The method is easy to implement and has some efficiency benefits. This is probably the method that is used most today. The following steps describe the steady state algorithm:

1. Initialize the population.
2. Randomly chose a subset of the population, the tournament size.
3. Evaluate the fitness of the participants in the tournament.
4. Select the winner or winners in the tournament i.e. the fittest individuals.
5. Apply genetic operators to the winner/winners.
6. Replace the losers in the competition with the individuals resulting from step 5.
7. Repeat step 2-6 until some termination criteria is fulfilled.
8. Present the best individual.

The tournament size is often chosen to be quite small. For example in Discipulus, which is the GP-engine used in this paper, the tournament size is 4.

In GP there are nowadays 3 common structures for representing individual programs namely tree, linear and graph based. The focus in this paper will be on tree and linear based. Graph based is basically only nodes connected by edges where the edges work as pointers between nodes and also indicating the flow of the program. There are different flavors of tree and linear representation. Tree can be of binary type or multidimensional and linear can be stack-based, register-based or machine code. Register-based and machine code is essentially the same; in both cases data is available in a small number of registers. Each instruction reads its data from registers and put its output to a register. This and the binary tree approach is the one that will be presented in this paper. A simple example here will work as a demonstration for how the different structures phenotypes look. Assume we have the polynomial \((A+B)*(C/D)\) and we want to represent it in a tree and a linear way the result of this is shown in figure 2.
A standard execution for a tree is to evaluate the leftmost node for which all inputs are available. For the linear version, which is a linear machine code, it is just instructions that are executed from the top to the bottom. The biggest benefit with the linear structure over the tree version is the speed of the execution. It works on the machine code level and there is no need for runtime compilation as it is when using higher level representations like in the tree structure.

2.1.2 History
GP is a relatively new member of the machine learning family. But as early as 1958 Friedberg\(^3\) attempted to solve simple problems by teaching a computer to write programs. Even thou there has been interesting progress in the area of GP it have not been possible to use in many applications until recently. GP is very computing intensive and there have not been fast enough computers until now to be able to solve problems inside reasonable timeframes.

GP is part of a family of computer simulation of evolution called evolutionary algorithms (EA). The main feature of EAs is that they try to mimic natural evolution. Perhaps the most famous member of EA is genetic algorithms (GA) which is the predecessor of GPs. Until not too long ago most efforts have been in the area of optimization rather than program induction. GA often is used as an optimization method where ordinary methods do not work very well. The original version of GA has two main characteristics: it uses a fixed length binary representation and uses crossover frequently. An example will show how GA works. Assume the value of \(x\) is looked for so that the function

\[
f(x) = x \ast \sin(5\pi \ast x) + 1, \quad x \in [-1, 2]
\]

get its maximum value. This is easy to do in a numerical way by hand but this will work as an example how GA works. A binary vector is used to represent a chromosome. The length of this vector depends on the required precision of the answer. If we would like an answer with five places after the decimal point, and because the domain of \(x\) has the length 3, we would

\(^3\)Bahnzaf, Nordin, Keller, Francone (1998)
need to divide that range into $3 \times 10^5$ equal size ranges. With this precision 19 bits is needed for representing $x$:

$$2^{18} < 3 \times 10^5 < 2^{19}$$

The mapping from the binary string $\{b_{18}b_{17} \ldots b_0\}$ to a real number $x$ goes through two steps:

$$\{b_{18}b_{17} \ldots b_0\}_2 = (\sum_{i=0}^{18} b_i 2^i)_{10} = x' ; \quad x = -1 + x' \times \frac{3}{2^{19}-1}$$

The first step is just a mapping from binary to a decimal format, and the second one puts $x$ into the desired range between $[-1, 2]$. After this representation is done a population is initialized and this is a simple step of randomizing bit vectors of length 19. This is simple because there is no combination of 0’s and 1’s that is not allowed here. When this is done the evolution process can begin. GAs uses crossover and mutation. The mutation operator goes through the bit string and flips the bits with a certain probability, and the crossover operator works on two individuals as shown below. The swap

$$c_1 = (00010|0001011100100011) \quad c'_1 = (00010|1101100110101000)$$

$$c_2 = (01001|11011001001101000) \quad c'_2 = (01001|00010111100100011)$$

point is located after the fifth bit and the content of the bit strings are swapped after that threshold. The chromosomes $c_1$ and $c_2$ results in the chromosomes $c'_1$ and $c'_2$ after the crossover operation. Both mutation and crossover operator gets invoked with a certain probability so if no one of them is used on an individual that one is passed to the next generation unaffected. This is called reproduction. An evolution function is used to evaluate the chromosomes and in this case a chromosome representing a value of $x$ is considered better if the target function gets a bigger value. One common way for the selection process is the following:

1. Calculate the fitness value $f_i$ for each chromosome $c_i$ ($i=1, \ldots, pop\_size$) where $pop\_size$ is equal to individuals in the population.
2. Pick out a chromosome to be copied to the next generation. Chromosome $c_i$ has probability $p_i = \frac{f_i}{\Sigma_i f_i}$ to be chosen. The denominator is the total fitness of the population so a fitter individual will have greater chance of being copied to next generation than a less fit.
3. Repeat step 2 $pop\_size$ times. This means that some chromosomes will be selected more than once and this is in line with natural evolution where more fit individuals have greater chance of survival.
4. Apply genetic operators on the chosen individuals in the new generation with some probability.
5. Repeat step 1-4 until some termination criteria is fulfilled. It could be that a certain amount of generations has passed or a value of the target function is reached.

GAs and GPs have many similarities which is natural when GA is a predecessor to GP. But they are also very different. The main difference between GPs and GAs is the representation of the solution. In GP the individuals in the population are not fixed-length character strings that encode possible solutions to the problem instead they are programs that when executed are the candidate solutions to the problem. In GAs it can be very hard to come up with a good representation of the problem, i.e. to find a proper bit string representation for a certain problem. In GP this is not really a problem because the system itself creates programs that represent the problem.

GP has many predecessors and the most important ones are listed in table 1. Smith’s work in 1980 where he developed a variant of Holland’s classifier system and each chromosome was a program of variable length may have been the first system inducing complete programs. Two researchers namely Koza and Cramer suggested that a tree structure should be used to represent each individual. In 1992 Koza came out with a book about genetic program which serves as a milestone in GP history.

<table>
<thead>
<tr>
<th>Year</th>
<th>Inventor</th>
<th>Technique</th>
<th>Individual</th>
</tr>
</thead>
<tbody>
<tr>
<td>1958</td>
<td>Friedberg</td>
<td>Learning machine</td>
<td>Virtual assembler</td>
</tr>
<tr>
<td>1959</td>
<td>Samuel</td>
<td>Mathematics</td>
<td>Polynomial</td>
</tr>
<tr>
<td>1965</td>
<td>Fogel, Owens, Walsh</td>
<td>Evolutionary programming</td>
<td>automaton</td>
</tr>
<tr>
<td>1965</td>
<td>Rechenberg, Schwafel</td>
<td>Evolutionary strategies</td>
<td>Real-numbered vector</td>
</tr>
<tr>
<td>1975</td>
<td>Holland</td>
<td>Genetic algorithms</td>
<td>Fixed-size bit string</td>
</tr>
<tr>
<td>1978</td>
<td>Holland, Reitmann</td>
<td>Genetic classifier system</td>
<td>Rules</td>
</tr>
<tr>
<td>1980</td>
<td>Smith</td>
<td>Early genetic programming</td>
<td>Var-size bit string</td>
</tr>
<tr>
<td>1985</td>
<td>Cramer</td>
<td>Early genetic programming</td>
<td>Tree</td>
</tr>
<tr>
<td>1986</td>
<td>Hickin</td>
<td>Early genetic programming</td>
<td>LISP</td>
</tr>
<tr>
<td>1987</td>
<td>Fujiki, Dickinson</td>
<td>Early genetic programming</td>
<td>LISP</td>
</tr>
<tr>
<td>1987</td>
<td>Dickmans, Schmidhuber, Winklhofer</td>
<td>Early genetic programming</td>
<td>Assembler</td>
</tr>
<tr>
<td>1992</td>
<td>Koza</td>
<td>Genetic programming</td>
<td>Tree</td>
</tr>
</tbody>
</table>

Table 1: Development of evolutionary programming.

GP has up to today been successfully applied to different problems such automatic design, pattern recognition, data mining, robotic control, bioinformatics and picture generation.
2.1.3 Theory

GP can be seen as a search process where an evolution process run on a population of individuals corresponds to a search through a space of potential solutions. Because individuals in a GP population don’t have a fixed size but can vary in size this means the search space is basically infinite. Program search spaces are usually assumed to be neither continuous nor differentiable which means classical optimization methods will not be able to solve the problems. So some kind of heuristic or stochastic search technique, like GP, must be used.

One way to visualize the search space is to use a fitness landscape. In a GA problem if the individuals can be visualized in two dimensions a plot in three dimensions can serve as a good help for understanding the basic concept in the search process. Figure 3 shows a very simple fitness landscape.

![Fitness landscape](image)

Figure 3: Fitness landscape

The task of all search techniques is to look for the best solution in the search space which here is represented by the fitness landscape. If the problem is a max-problem the highest peak is the target of the search. In a minimum problem the lowest valley is what you are looking for. In a greedy search algorithm which only considers adjacent points the local gradient will take the search to nearest peak, and as can be seen in figure 3 there is no guarantee that this is the global optimum. The peak that will be reached is dependent on the starting position which is a well-known problem in linear programming. A good comparison can be made with a mountain climber that is shot out of an airplane at some random place over the Himalaya. When the person lands he or she immediately starts to climb the nearest peak and at some point the peak will be reached, however there is no guarantee it is Mount Everest that has been climbed. In some sense in GA one can think of it as many mountain climbers jump out of the plan and start to climb different mountains. Individuals that have found reasonable high mountains (high fitness) are rewarded by having children which can continue the search for the top. Unfortunately it is much harder to use this metaphor for GP because of its dynamic nature and possibility to alter its structure and size.
A more theoretic approach to try to understand how the search in GAs and GPs is based on the idea of dividing the search space into subspaces called schemata. In the context of GA that is working on binary strings there is a set of three different symbols \{0, 1, #\} which the schema can be built on. The # stands for “don’t care” and can be replaced by a 0 or a 1. So a schema like 0#1# represents the bit strings 0010, 0011, 0110 and 0111, and the other way around 0010, 0011, 0110 and 0111 matches schemata 0#1#. The number of non-# in the string is called the order \(O(S)\) of a schema \(S\). The distance between the furthest two non-# symbols is called the defining length \(\delta(S)\) of schema \(S\). Another important property of a schema \(S\) is its fitness at time \(t\), \(eval(S,t)\). If there are \(p\) individuals \((v_1, ..., v_p)\) in the population at time \(t\) that is matched by schemata \(S\) then:

\[\text{eval}(S, t) = \sum_{i=1}^{p} \text{eval}(v_i)/p\]

Where \(\text{eval}(v_i)\) is the fitness of individual \(i\). As was explained in the part about the GA algorithm individuals are copied to a temporary generation with a certain probability based on its fitness so an individual may be copied zero or many times during this phase. After this selection step we expect to have \(\xi(S, t+1)\) strings matched by schemata \(S\).

\[
\xi(S, t + 1) = \xi(S, t) \times \text{population}_{\text{size}} \times \text{eval}(S, t)/F(t)
\]

[1]

It is quite easy to see why [1] is true based on three steps:

1. For an average string matched by schemata \(S\) the probability that it will be chosen for copying is equal too \(\text{eval}(S,t)/F(t)\), where \(F(t)\) is the total fitness of the population at time \(t\).
2. The number of strings matched by schemata \(S\) at time \(t\) is \(\xi(S, t)\).
3. The total amount of individuals in the population is equal to \(\text{population}_{\text{size}}\).

Formula [1] can be rewritten to [2] because \(\bar{F}(t) = F(t)/\text{population}_{\text{size}}\), where \(\bar{F}(t)\) is the average fitness in the population.

\[
\xi(S, t + 1) = \xi(S, t) \times \text{eval}(S, t)/\bar{F}(t)
\]

[2]

As can be seen in [2] a above-average schemata gets an increasing number of strings in the next generation and a below-average schemata gets a decreasing number of strings in the next generation. An average schemata stays on the same level. If we assume a schemata \(S\) remains above average by \(\varepsilon\% (\text{eval}(S, t) = \bar{F}(t) + \varepsilon * \bar{F}(t) )\) then we get:

\[
\xi(S, t) = \xi(S, 0)(1 + \varepsilon)^t
\]

Now an above-average schemata not only gets an increasing number of strings but get it at an exponentially rate.
Until now genetic operations haven’t been mentioned and without them there will be no change of the individuals but only a convergence of all individuals to the best schemata. Crossover will be looked at first. The defining length \( \delta(S) \) of the schemata \( S \) is important when discussing the crossover operator. The longer the \( \delta(S) \) is the bigger chance is it that the schemata will be “broken” during the crossover. A crossover point, swap point, is decided randomly and on a string with length \( m \) there are \( m-1 \) places where the crossover can occur. So the probability that schemata will survive the crossover is equal to:

\[
p_s(S) \geq 1 - p_c \frac{\delta(S)}{m - 1}
\]

The reason there is a bigger-than sign rather than an equal sign is because even if a crossover point is chosen between fixed points in the schemata there is a chance it won’t destroy the schemata. \( p_c \) is the probability that the crossover operator will get invoked, because if it isn’t the schemata will survive. The mutation operator works on each bit in the string with a probability \( p_m \) and flips the bits. The order \( O(s) \) is obviously important here because the higher the order is the greater probability that the schemata will not survive. The probability that a single bit will survive the mutation is \( 1-p_m \), so for the whole bit string the chance for survival is:

\[
p_s = (1 - p_m)^{O(S)}
\]

The combined effect of selection, crossover and mutation gives us the final formula for the expected numbers of strings matching schemata \( S \) in the next generation:

\[
\xi(S, t + 1) \geq \xi(S, t) \times \left( \frac{\text{eval}(S, t)}{\bar{f}(t)} \right) \times \left( 1 - p_c \frac{\delta(S)}{m - 1} \right) \times (1 - p_m)^{O(S)}
\]

The schema theorem of Holland \(^4\) described above addresses the question why the algorithm works. In essence it states that good schemata, which works as building blocks, tends to multiply exponentially and together with other good schemata form good solutions.

It is hard to transfer the schema theorem to GP because the representation in GP is much more complex with varying length and it allows genetic material to move from one place to another in the genotype. Many attempts have been made and the first out was Koza. He argues that a schema is a set of subtrees that somewhere contain one or many subtrees from a special schema defining set. For example if the schema defining set is the set of \( S \)-expression \( H=\{(- 6 \ x), \ (+ 5 \ 4)\} \) then all subtrees containing \( (- 6 \ x) \) or \( (+ 5 \ 4) \) are instances of \( H \). His reasoning suggests that the fact that GP crossover tends to preserve good schemata rather than destroying them is thanks to the reproduction factor, which creates additionally copies of an individual without changing them. Individuals that contain good schemata is likely to be more fit and therefore have bigger chance to be selected for reproduction. Good schemata will then be tested more and recombined more often than worse schemata. This

\(^4\) Michalewicz (1996)
process results in the combination of smaller but good schemata into bigger schemata and finally good solutions. There have been a lot of other theories about GP and all seem to have in common that they have some restrictions or some other flaws. Ultimately it is a very hard area for making any general proofs.

Many researchers nowadays questions the validity of the schema theorem but it still serves as a good mathematical based analysis of the genetic operators. Many scientists have left the concept of schemata and their focus lies nowadays on so called Markov chain models.5

2.2 BASIC CONCEPTS
In GP there are many expressions and to be able to understand a paper about GP the reader must become familiar with some of the fundamental concepts. In this part some of basic concepts are discussed.

2.2.1 Population
During a GP-run there are a certain number of individuals on which the evolution process takes place. These individuals constitute the population. The size of the population will be a parameter of the run. Before a GP-run can take place an initial population has to be created. The individuals are generated with an initialization algorithm. For tree structures there is two common methods called full and growth. One parameter that has to be decided is the maximum depth of the tree. The depth of a node is the minimal number of nodes that must be traversed to get from the root node of the tree to the selected node. With the grow method the initial trees are created by randomly choosing the nodes from both the function and terminal sets. Once a branch contains a terminal that branch has ended. With this initialization the trees will most likely get an irregular shape. On the other hand if the full method is used the tree gets a regular shape because terminals are only chosen when the node is at the maximum depth.

2.2.2 Terminal set
The terminal set consists of the inputs and the constants to the GP-program( and sometimes also zero-argument functions). In other words the members of the terminal set all have an arity of zero. If a tree structure is used to represent the programs all leafs of the tree will be part of the terminal set. As an example if you are using the GP for solving a regression problem where the polynomial you are looking for is \( x_1^2 + x_2^2 + 3 \) then one row in your datafile might look like this:

2 6 43

The first column represents input \( x_1 \) and the second input \( x_2 \) and the third is the target value. If the GP is to come up with a perfect solution it needs to incorporate a constant namely three here to be able to solve the regression problem. So both the constant and the input variables \( x_1 \) and \( x_2 \) are part of the terminal set.

5 Langdon, Poli (2001)
2.2.3 Function set
The function set consists of all the functions, operators and statements available to the GP system. A couple of example is

- Boolean functions (And, Or, Not, Xor)
- Arithmetic functions (Plus, Minus, Multiply, Divide)
- Subroutines

The subroutines used are specific to the application. For example if you are trying to evolve a soccer application you might have implemented subroutines like shoot or pass.

The task of choosing which functions that are to be included in the function set is not trivial. One might be tempted to include every instruction there is because then you can’t accidentally leave an instruction out that is necessary for solving the problem. The problem with this approach is that the search space gets much bigger and the time it takes to come up with a good solution can be substantially longer. On the other side if only addition is included you won’t be able to solve any interesting problems. A starting set with Boolean functions and Arithmetic functions is often a good idea because you can solve a huge amount of problems with this set.\(^6\)

A perfect tree based solution to the polynomial mentioned above \(x_1^2 + x_2^2 + 3\) is shown in figure 4. Here the tree consists of 9 nodes where 4 are members of the functional set and 5 are members of the terminal set.

---
\(^6\) Bahnzaf, Nordin, Keller, Francone (1998)
2.2.4 Fitness function

The fitness function is the mechanism that is used for deciding how fit a certain individual is. The results of the fitness calculations serves as a guideline for the learning algorithm when choosing which individuals that is more likely to be chosen for GP operations, i.e. an individual with a better fitness value is more likely to be chosen than a individual with worse fitness value. The fitness value is calculated on the training set and during a run the fitness value should improve until a better value can’t be found. The last part is not entirely true because usually time is a restrictive factor during a run so often a run will be terminated if the fitness value hasn’t improved after a certain amount of generations. If the run had been allowed to run for more generation it is possible that a fitter individual had been found. If an individual with no fitness error at all is found during the run the problem is considered solved and the run is aborted.

The importance of a well written fitness function can’t be stressed enough because it is the key to a successful evolution. For symbolic regression problems it is usually a pretty straightforward process to write a good fitness function. A common way is to take the sum of the absolute value of the difference between the output from the individual and the target output over all fitness cases in the training set. If the target output of the $i$th fitness case is $t_i$ and the output of the GP generated program’s corresponding value is $p_i$ then the formula for the fitness value for program $p$ with $n$ fitness cases is:

$$f_p = \int_{i=1}^{n} |p_i - t_i|$$

The approach above is a linear measurement and the fitness value gets better the closer to zero it is. An alternative is to use the square of the difference instead which may give better search results in some cases. With the square approach values that are far from the desired output gets amplified and values below one gets dampened. There are many situations when the fitness function has to be made in a more custom manner such as:

- Financial applications, i.e. when you want to create a model that generate maximum profit over a certain time period.
- Artificial intelligence, i.e. when a robot is supposed to learn to move.

2.3 GENETIC OPERATORS

The genetic operators are fundamental in GP. They work in a way that will remind of their counterparts in natural evolution. In this section the common genetic operators are discussed.

2.3.1 Crossover

In most GP systems crossover is the dominating search operator. The reason for this is that GP crossover tries to mimic the process of sexual reproduction which in nature obviously has worked quite well. Crossover combines material of two parents by swapping some part of each parent’s code with the other. An example of a simple crossover operation is shown in
figure 5. One node in each tree is chosen randomly and the subtrees of the chosen nodes are switched between the trees.

![Figure 5: A simple crossover operation. a) Individual 1 before crossover. b) Individual 2 before crossover. c) Individual 1 after crossover. d) Individual 2 after crossover.](image)

Linear crossover works in a similar way and swaps a segment of instructions between the individuals rather than subtrees.

As stated in the theory part of this paper it has been argued that GP populations contain building blocks called schemata. Good building blocks improve the performance of individuals that include them and therefore they are more likely to be selected for genetic operation and their genetic material to be spread to the population. There is a significant difference between crossover in nature and in GP in terms of viable offspring. In nature most crossover events are successful and result in viable offspring. This is not the case in GP where around 75% of the crossover events over a whole run result in offspring that have less than half the fitness of their parents. In nature this would often mean that the offspring would die. So why is this the case in crossover in GP? An example will help with the explanation here. Figure 6 shows an individual in a population.

![Figure 6: A tree-based individual.](image)

---

7 Bahnzaf, Nordin, Keller, Francone (1998)
If the blue nodes constitute a good building block the probability that this block will get destroyed by crossover is 3/12 or 25% because there are 12 nodes in this individual that is available for crossover (the root node is usually not available as a crossover point). Let’s assume that crossover finds a new good building block, the orange nodes, and combine this one with the former good block creating a bigger building block including nodes 2-8. Now the probability that this block will get disrupted by crossover is 7/12 or around 58%. As more and more smaller building blocks get assembled into bigger blocks the whole structure becomes more prone to destructive crossover. The crossover operator is a two-edged sword with both beneficial and destructive crossover. Standard GP crossover is unconstrained and differs a lot from crossover in nature where there is strict rules where the crossover can occur and usually only resulting in minor changes. There are many different techniques in GP to try to duplicate natural evolution more closely, some described in the part about Discipulus.

2.3.2 Mutation
The mutation operator only works on a single individual at a time. Usually the mutation operator gets invoked with a low probability on the result of the crossover operator or if the crossover operator is not invoked the reproduced individual. There are many different types of mutations operations today in GP, one of the more common in tree GP is to randomly pick a node in the tree and replace the existing subtree starting in that node with a randomly generated subtree. Some other examples are listed in table 2.

<table>
<thead>
<tr>
<th>Mutation operator name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Point mutation</td>
<td>Single node exchanged against random node of same class</td>
</tr>
<tr>
<td>Permutation</td>
<td>Arguments of a node permuted</td>
</tr>
<tr>
<td>Hoist</td>
<td>New individual generated from subtree</td>
</tr>
<tr>
<td>Expansion mutation</td>
<td>Terminal exchanged against random</td>
</tr>
<tr>
<td>Collapse subtree mutation</td>
<td>Subtree exchanged against random terminal</td>
</tr>
<tr>
<td>Gene multiplication</td>
<td>Subtree substituted for random terminal</td>
</tr>
</tbody>
</table>

Table 2: Genetic mutation operators.

The mutation operator for linear GP works on the instruction level. First an instruction is chosen randomly and after that the mutation is performed on that instruction. The type of change usually is one of the listed above:

- The register used can be randomly changed to another register in the register set.
- The operator in the instruction may be changed to another operator in the function set.
- A constant used may be changed to another constant in the allowed range.

2.3.3 Reproduction
The reproduction operator is very simple. It is a copy of an individual that is copied into the next generation without being subject to mutation or crossover. The good building blocks
stay intact when the reproduction operator is invoked and a good building block therefore will have more chances to find beneficial crossover.

2.4 INTRONS

Introns are part of the code in an individual that has no effect of the individual’s performance. Examples of introns are:

- \( x = x + 0 \)
- \( y = y \times 1 \)
- \( \text{if}(5<1) \text{ then } \{ ... \} , \text{ the code in the block will never be reached} \)

P.J Angeline noted that this extra code seemed to emerge spontaneously from the process of evolution as a result of the variable length of GP structures and that this property may be important for successful evolution. He was the first to make the association with introns in nature. Even thou they may be very different both seem to have the feature of constituting a big part of their environment and have no actual effect of their owner’s behavior. Studies indicate\(^8\) that in the early and middle phase of a GP run introns stands for 40% - 60% of all code. In later parts of a GP run another interesting phenomena occurs which is called bloat. At this phase introns tends to grow exponentially and to comprise almost all code in the entire population. The only limit for the growth is the maximum size of the program, which is a parameter of the run. When this exponential growth process takes place the GP run usually is unable to undergo any further evolution and if the run is allowed to continue the chances are low that an improvement will occur.

2.4.1 Why does introns exist in GP

The introns by themselves may seem pointless but they play an important role when the fate of the individual’s offspring is decided. For this discussion another type of fitness is used called effective fitness. The effective fitness does not only consider the individual’s fitness but also the likely fitness of the offspring. With this approach the ability of the children to be highly fit is as important as the parents so the good genes can continue to exist in the population. It does no good if the parents are highly fit but their offspring has poor fitness and will get discarded fast. As described in the part about crossover the normal effect of crossover is that the children are much less fit than its parents. Introns help prevent this fact. The common belief today is that introns emerge in order to “protect” the good parts of the code from destructive crossover. When introns appear in the code the possible spots where crossover can take place increases which in turn mean that the probability that the good building blocks will be broken decreases. So in terms of effective fitness the better the parents is in protecting its children from destructive operations the higher effective fitness it has. It is important to note that destructive crossover is not always bad nor is children with worse fitness rather it is a phenomenon that is part of evolution and not fully understood and this is just an explanation of why they do occur. If the GP system was forced to only

\(^8\) Bahnzaf, Nordin, Keller, Francone (1998)
accept children with higher fitness level than its parents then it would basically be a hill climbing algorithm and much of the power with GP would be lost. When bloat turns in and the introns starts to increase exponentially it is believed that this is the GP:s way to try to protect the individual from any change at all. The reason for this is at the end of a run there is very hard for an individual to do any improvements because they should be close to their best value so instead the focus turns to preventing any destructive operation from destroying the good solutions already found. Finally it should be noted that also mutation is usually destructive when applied in GP.

2.5 DISCIPULUS

Discipulus is a GP software package that can solve regression and classification problems. It is one of the few, or possibly the only, commercial GP product available today that can offer evolution at a very high speed. Discipulus works on machine code level and the creators claims it to show speed gains 60 to 200 times over other designs. TSL is built upon Discipulus or rather Discipulus is an integrated part of TSL. During my work I have had access to both TSL and a standalone version of Discipulus which has been very useful for understanding Discipulus when its features are kind of restricted in TSL.

To run Discipulus with standard settings is a pretty straightforward procedure. You just choose which data files you want to use and then create a new project. During evolution Discipulus can use up to eight computation variables which are held in floating point registers called f[0],f[1],...,f[7]. A simple example will show how the results in Discipulus are presented. Assume we have a regression problem and our data exactly match the polynomial \(a^3 + b^2 - c\) then two rows in our data file can have the following appearance:

\[
\begin{array}{cccc}
4 & 5 & 10 & 79 \\
-3 & 16 & 4 & 225 \\
\end{array}
\]

This will be a very simple problem for Discipulus to solve and will only take a few seconds. Obviously you will need to have more than 2 rows in your data file otherwise there are many other possible solutions to the problem. The result presented in Discipulus to this particular problem can look like this:

L0: f[0]+=v[1]  
L1: f[0]*=f[0]  
L2: f[0]=v[2]  
L3: f[1]+=f[0]  
L4: f[0]=f[0]  
L5: f[0]+=v[0]  
L6: f[0]*=v[0]  
L7: f[0]*=v[0]  
L8: f[0]+=f[1]

\(^a\)http://www.rmltech.com/
The input variables from the data file is called \( v[0], v[1] \) and \( v[2] \) and there are three here because it was three in the data file. \( v[0] \) corresponds to the left most input variable, \( v[1] \) to the second and so on. The right most input in the data file is the desired output. The register \( f[0] \) has two roles in Discipulus. To begin with it serves as a temporary computation variable but it also holds the output of a program when it has finished executing. In this example there was an exact solution, i.e. the fitness error was zero. In many problems there is no perfect solution so the system has to aim for getting the best solution possible. The code in the example above consists of 9 lines and is easy to follow. This is not always the case thou, usually the code is filled with introns. For this simple example the code originally consisted of 24 lines but fortunately Discipulus has a feature in it that can remove introns from the code which can be very tedious to do by hand. The final code can be saved as C, Java or Assembler code which means that you don’t really have to care about if it contains introns or not unless you want to analyze the actual code. The result will be the same regardless of the presence of introns or not.

A common approach in GP, and also in Discipulus, is to divide the data into three different sets called the training, validation and applied sets. A good start can be to make the sets the same size. The usage of these sets differs slightly between different GP systems. In Discipulus the programs are built based on the training set and then evaluated on the validation set and the resulting fitness value is the mean of the fitness of training and validation sets. This means that programs that work good both in the training data and validation data are favored. The applied set is not used in the evolution at all but works as an out of sample test to see how good the model works on data that it has no prior knowledge about. The best way to decide how good the resulting program really is is to see how well it performs on out of sample data. If the fitness error of the out of sample data is similar to the error in training and validation you most likely have a good model representing the data. On the other hand if there is a big difference between in sample and out of sample data the system is over fitted to the data it was built on. There are some ways to try to eliminate over fitting and Discipulus also have some features integrated for avoiding it.

- Get more data if that is possible. This is the easiest way for improving performance.
- Reduce the number of inputs. When a run is over in Discipulus an input impact summary is presented. In this you can see the impact of the variables in the best programs. From this it is possible to remove the least important variables.
- Reduce the target size of the program. This is a parameter in Discipulus and a shorter program has to be more general and hopefully will work better out of sample.
- The usage of parsimony pressure. Parsimony pressure is a term used to refer to techniques that tend to make the evolved programs shorter. It is a parameter in Discipulus if you want to use it or not. It is applied during a tournament and compares the two winners of a tournament and if they are inside a certain threshold of each other the shortest of the program is considered the winner. The proportion of tournament it will be applied to and the size of the threshold is parameters that
are up to the user to set. Usually a very small threshold should be used, like 1 %, otherwise the solutions from Discipulus may be short but also bad.  

Discipulus uses two different kinds of crossover techniques. Homologous crossover tries to work in a way more similar to how crossover works in nature. Programs are lined up next to each other and the crossover occurs by exchanging groups of contiguous instruction blocks. The blocks are of the same length and from the same position in both programs so it does not affect the length of the programs. With non-homologous crossover there is no concern taken to where the crossover occur or the length of the changed blocks. Non-homologous crossover must be used if the GP shall be able to evolve program of different size.

At last it should be clear that if no actual pattern exist in the data then it is not possible for the GP system to come with a good solution regardless of how advanced it is.

2.5.1 Fitness calculation
Discipulus is sold in different versions where the more expensive ones support writing your own custom fitness functions in addition to the standard fitness calculations integrated in Discipulus. The standard fitness error calculation can be used in a linear or square mode which is a parameter of the run. The linear fitness errors is calculated by adding the errors for the individual over all fitness cases (rows in the data file) and then divide it by the number of fitness cases. The square linear fitness error is calculated in the same manner with the difference that each fitness case error is squared.

The possibility of writing your own fitness function can be necessary to have when the problem under consideration can’t be handled in a good way by the standard fitness calculations offered by Discipulus. For example if you want to create a model that trades automatically in the financial markets and its performance is measured in how much money it can generate you may want to create your own fitness function that takes consideration to net profit and maximum drawdown and other things. There are two different custom fitness functions available in Discipulus, a vector and a pointer version. The custom fitness function is used by creating a dynamic-link library (DLL) which you can call from Discipulus. The pointer version is the one I have worked with during this work. The reason for this is that it is more powerful and gives you direct access to the evolved program via a pointer that is passed from Discipulus to the DLL. The DLL is responsible for handling all training and validation data and for calculating the fitness error and return them to Discipulus.

2.5.2 Instructions
In Discipulus you can choose which instructions you want to include in the evolution. Depending on the nature of the problem some of the instructions might not be necessary for the solution. If all instructions is used the downside is that the evolution takes longer time, on the other hand if an instruction that is needed is left out it can be hard to get a good result. The group of instructions that is available in Discipulus is the following:

---

10 Discipulus manual
• The addition group contains three different instructions for addition. The difference between the instructions is how they use the registers.
• The arithmetic group has 4 different instructions which include absolute value, change sign, a scaling instruction and finally square root.
• The comparison group only include one instruction that compares the values in two registers and sets a flag to one if the first register contains a value that is smaller than the value in the second register otherwise it sets the flag to zero.
• The condition group contains four instructions for handling conditional instructions such as skipping instructions if a flag is set.
• The data transfer group only has one instruction which swaps values between different registers. This is important for temporary storage of variables.
• The division group contains four instructions. Three are basic division instructions that use registers in different ways and the last calculates the remainder.
• The exponential group has one instruction that calculates two raised to a power held in a specific register and that result is subtracted with one and saved in a register.
• The multiplication group consists of 3 instructions and similar to the addition group the difference between them is how they use the registers.
• The rotate stack group contains 2 instructions where the first decrements the FPU stack pointer with one and the other increments it with one.
• The subtraction group is also similar to the addition group and contains three instructions for subtraction where the difference lays in how they use the registers.
• The trigonometric group has 2 instructions in it, one for cosine and one for sine.

You don’t have to include all instructions in a group instead you can choose which one you want to use. The process of choosing which instructions to include is not trivial but rather a trial and error process if there is no knowledge about the input data. Some instructions should never be left out like addition and multiplication because they are used by Discipulus to get the inputs into the FPU registers.

2.5.3 Other settings of interest
There are a couple of other settings in Discipulus that are worth mentioning. One is the concept of demes. In biology it is believed that genetic diversity is enhanced when same species are isolated from each other geographically. For example a part of a population of a certain species may be isolated on an island located somewhere in the ocean. The blending of the genetic material of the inhabitants of this island is isolated to that group. These isolated groups are called demes. The main purpose with demes in GP is to keep genetic diversity in the population and avoid getting stuck in a local minimum early in the evolution. There are three run parameters for deme control in Discipulus. The first one sets how many demes there will be in the population. The second one sets the crossover rate between different demes, i.e. that an individual from one deme are engaged in crossover with an individual from another deme. The third one sets the migration rate between different demes. Normally an offspring is kept in the same deme as its parents but this parameter sets
how many that will migrate to other demes. The two last parameters should be kept at a very low percentage or the purpose with demes disappears.

Another run parameter is the usage of Dynamic subset selection (DSS). DSS helps with speeding up the evolution. When DSS is used the fitness error is not calculated over all fitness cases but instead only on a part of them. The fitness cases used are changed during the run. The size of the subsets and how the change of the fitness cases is made during the run are other parameters but they won’t be described in more detail here.

You can also set the amount of homologous crossover that will be used and therefore indirectly the amount of non-homologous crossover used. There are also settings for the mutation operator. There are three different alternatives where you can set their individual significance; all three just need to sum up to 100%.

- Block mutation rate - Programs in Discipulus have their instructions inside of instruction blocks that are 32 bits in length. This parameter sets what percentage of the mutation operations that replaces an entire instruction block with a new randomly generated instruction block.
- Instruction mutation rate – This parameter sets the percent of mutation operations that result in a single instruction being replaced by a new, randomly chosen instruction of the same length.
- Instruction data mutation rate - This parameter determines the probability that the mutation operator will change the inputs or the constants to which an instruction refers to another temporary computation variable, input, or constant.

Finally there are different run termination criteria that can be set. A run can be terminated after a specific number of generations or after a certain number of generations without improvements.
3 FINANCIAL APPLICATION

This chapter is divided in two parts and the first presents the concept of automated trading. The second part is about TSL which is a software package that creates trading models with the help of genetic programming.

3.1 AUTOMATED TRADING

The usage of computers is absolutely vital today in the financial markets. Traders send orders to their brokers by computerized systems or by phone and the brokers in turn have access to the exchange with their systems. Private persons don’t have direct access to the exchange so they have to go through their broker. Traditionally trades have some analysis behind them and if someone thinks the share price of a particular stock will go up they will go long in that stock. On the other hand if they expect a fall in the share price they will go short. The recent years automated trading has become more popular.

Automated trading can have a different meaning in different contexts but here it means the concept of computer software that trade by itself without necessary interference from humans. Automated trading differs from traditional trading in the sense that it makes no analysis of the underlying asset. It does not have any knowledge about if the company is undervalued in the case the underlying asset is a stock or if the future looks particular good for this specific company rather it acts on certain trade signals. Trade signal in this context means that a certain criteria is met and therefore generates a sell or buy signal. A trade signal can be that the share price goes below a certain threshold and then indicating that now is a good time to buy or that the share price has gone up a certain amount of days in a row and therefore indicating an upward trend. This is closely related to a field called technical analyses where the goal is to predict prices with the help of historical data. Technical analysis builds on the belief that there are patterns in price movements so if you study historical data you can foretell future movements. Technical analysis stands in contrast to the so called efficient market hypotheses (EMH) which many financial theories are based upon. EMH says that financial markets are efficient and the prices reflect all known information. This means that is not possible to beat the market consistently with information already known to the market. EMH comes in three different forms:

- Weak efficiency states that excess returns can’t be made with the help of historical data so technical analysis won’t work
- Semi-strong efficiency states that no excess return can be made with the help of public information like reports from companies. For this to be true the information made public must immediately be reflected in the prices.
- Strong efficiency states that excess returns can’t even be made with inside information. In this form all information both public and private are reflected in the share price.
Whether EMH is true or not has been widely debated over the years. At least the strong form has been empirically shown not to be true in most cases. A common belief is that psychological effects among humans are a cause to imperfections in financial markets. Humans don’t always react rationally.

3.2 TSL

TSL is a software package that creates trading models with the help of genetic programming. A trading model is a logical set of instructions that tell the trader when to sell or buy a particular instrument. The trading model can be used for stocks, future contracts or other derivatives of interest. Traditionally trading models has been created by hand by studying historical data and from this try to find pattern in the data. This is a very time consuming process and often the resulting model has poor performance due to over fitting. TSL can create trading models in a matter of minutes hence offering a huge efficiency gain.

3.2.1 Preprocessing

Before a run can start in TSL you need to get data that the models are going to be built upon. There are a couple of ways of getting this data. During my work I have used TradeStation for this. TradeStation is a technical analysis software that is used for analyzing and trading the financial markets. It has its own programming language called EasyLanguage that can be used to write your own models and create indicators and many other things. Indicators assist in forecasting a market’s direction and assist traders in making decisions related to entering, exiting, and/or staying in or out of the market. Indicators can be of huge help when analyzing things like trends, trend reversals, and volatility. There is a big amount of predefined indicators in TradeStation. The indicators works as input variables to the GP engine and in TSL 56 input variables are used. Unfortunately it is impossible to tell which indicators TSL are using because the calculations with which the indicators are calculated with are hidden within a DLL-file. A maximum of 60000 rows in the data file can be used in TSL. A row in the data file corresponds to a bar in TradeStation. The appearance of the bars can be seen in figure 7.

---

12 http://www.tradestation.com
The little horizontal line of the left side of the bar is that bar’s opening price, the height of the bar is the range of that bars duration so the bottom of the bar is the lowest price and the top the highest price and finally the right little horizontal line is that bar’s closing price. On each bar all indicators are recalculated so for example a moving average that is based on the ten last bars will be recalculated on each bar. The instrument I will look on in the example in the next part is Standard & Poor’s 500 E-mini contract which is a stock market index futures contract traded on Chicago Mercantile Exchange (CME). This is one of most highly traded futures contract in the world\(^{13}\) and the trading is fully electronic. When you preprocess data from financial markets you usually want to get rid of anomalies in the data such as huge price movement that is related to some political event or a natural disaster. Those are one time occurrences and you don’t want to build models based on those. If you are looking on a stock rather than a future contract you might want to adjust for dividends as well. The problem with TSL is that you don’t really have any knowledge about the preprocessing instead you just need to accept it as it is. There are a couple of integrated “preprocessors” in TSL which you can use if the instrument you are looking on seems to have certain characteristics which also can be called data movement personality such as trending or counter trending or a mix of the two.

### 3.2.2 Settings

TSL has a lot of settings that can be set by the user and some of them will be described in this area. For starters TSL has support for creating models for single instruments like futures or stocks but also options, pairs and portfolios. During my work I have only worked with single instrument and mainly different futures contract. There are many different fitness functions available in TSL and they are shown in picture 8.

---

\(^{13}\) http://www.cme.com/trading/prd/equity/emini-sp500.html
In Discipulus a lower fitness value is better and therefore it is necessary to invert the fitness functions final value otherwise a high net profit would be considered bad for example. When using certain fitness functions you have to force the system to do a minimum number of trades otherwise the system will most likely come up with trivial solutions with no trades at all or perhaps one profitable trade. An example of such a fitness function is [Net Profit/Max Drawdown]. As also seen in the picture it is possible to choose the trade direction. You can choose to go both long and short or just one of them. The big amount of different fitness functions gives the user the freedom to choose a fitness function that suits him or her well. Theoretically a plain net profit fitness function can be used for maximum possible gain, a fitness function with max drawdown or standard deviation of the return in the denominator can be used for having a more linear return and Sharpe ratio can be used to characterize how well the return compensates for the risk taken.

When fitness function and trade direction and possibly a minimum number of trades are chosen the next step is to choose entry and exit rules on the market. This is basically the rules of how the trades will be made. The different alternatives are shown if figure 9. Alternatives that build on “This Bar’s Close” are not very good to use when working with TradeStation. The reason for this is how TradeStation handles orders. When TradeStation registers that the bar is closing it tries to send an order to the market. But at this time the market may have closed and won’t accept the order and even if it is not closed the order won’t be executed until the next bar anyway. So it does not make much sense to use this alternative when working with TradeStation. Enter on Limit means the order is of the type “limit-order” which means that you buy or sell at a specified price or better. If you sell at the
market price you will get the price someone is willing to pay at the moment but when you put a limit buy order you won’t buy until someone is willing to sell for that price.

One of the benefits with limit orders is that you protect yourself against slippage costs. Slippage is the difference between the expected price of a trade, and the price the trade actually executes at. If you want to sell 100 shares and the current market price is 50 SEK for the share you might not end up selling all shares for 50 SEK (or you might end up selling no one for 50 SEK). If you sell at a market order and there is only buyers for 50 stocks at 50 SEK the rest of the order will be executed at a lower price. If the last 50 shares are sold for 48 SEK you have effectively sold the shares for 49 SEK a piece. You can get the same effect even if you are selling only one contract due to the fact you will be put last in the queue of people that have sent their order before you at the same price. So if there are not enough buyers at that particular price the order will be executed at a lower price. The rules at the end of the list with a [B] in front of them are “beta-rules” and haven’t been used that much during my work.

There are other settings of interest in TSL and some of them are shown in figure 10. The meaning of the parameters shown in figure 10 is described in the following part.
Stop loss exits is a way to limit an investor's loss on a position. An order to buy or sell a position is sent when its price exceed a predefined threshold. Within this area in TSL you can set the maximum loss you can do before you sell or cover your short position. There are four different options available here:

- **No Stops** – no protection is used against losses.
- **GP Stop Exit** – the GP engine tries to come up with the most efficient stop loss levels.
- **Fixed $ Stop** – a certain dollar amount is set to describe the biggest allowed lose.
- **Fixed % Stop** – a fixed percentage of the original investment is used as the stop loss.

Profit targets is similar to the stop loss but instead for exiting your position after a certain loss an exit occurs after a certain amount of profit is made. You can choose if the profit target will be based on an absolute dollar amount or on percentage basis. There is also an option for letting the GP try to come up with optimal exit levels and then exit on a limit order.

Risk and size settings gives the opportunity to create trading systems that are more dynamic in their nature and the number of contracts traded is based on the volatility of the market at the time that the trade is executed. There are two options in this category one is based on classic standard deviation and the other is based on Average True Range (ATR). The True Range indicator is calculated from information from the bars and is equal to the greatest of the following:

- current bar’s high less the current bar’s low
- the absolute value of the current bar’s high less the previous bar’s close
- the absolute value of the current bar’s low less the previous bar’s close

Average true range is a moving average of true ranges. An instrument with a high level of volatility will have a higher ATR than one with a smaller degree of volatility.

The n bar exit setting is used if you want to force to exit from your position after n days. The money management settings are used if you want the GP system to change the number of contracts held during the run so it can try to minimize the error against the fitness function. With a fitness function like “net profit” this will only mean the trivial case that it uses the maximum allowed number of contracts at all times. For a fitness function like “net profit/sqrt(MaxDrawdown)” it may turn out that the number of contracts held are changing over time.

As seen to the right in figure 7 it is possible to set the percentage of out of sample data used in the run. With this setting set to 20 % means that the models will be built upon 80 % of the available data and the remaining 20 % will be used for OOS. In TSL the training and validation data is the same so the “Val Used”-parameter shouldn’t be changed. The next available setting is the “R.T.S and C ,$” which is the slippage and commission cost for a round turn. A round turn includes both an entry and an exit. This is a very important setting and has severe
consequences on the generated models which will be described in more detail later. The risk free interest rate is needed for certain fitness function calculations, i.e. Sharpe ratio. The account size is the available money for the trades. In the case of futures contracts you don’t have to pay anything when you get the contract. Futures contracts are marked to market\(^{14}\) which means profits and losses are accounted each day and you will need to have a certain margin on your account so you will be able to cover the possible loss. The size of this margin depends on the contract’s size and the deal you have with your broker. So in the case of futures the account size can be thought of as the margin.

3.2.3 Example run

In this example I will use data from 20041101 to 20071101 with 60 minutes bars which generated 18132 rows. The reason I only took data until 2007 instead until today is that it gives me the benefit of being able to test the generated model from 20071101 to a future date for which the data is known and see how well it actually would have performed if it had traded “live”. The settings I am using for this example run are:

- Fitness function – Net profit
- Entry exit tactic – Next bar’s open
- Trade direction – Both long and short
- OOS used -20 % OOS data used
- Slippage and commission cost per round turn – 20 $

As described earlier the instrument I am looking at is the Standard & Poor’s 500 E-mini futures contract. The result after 5 minutes can be seen in figure 11.

---

**Figure 11: Result after 5 minutes.**

\(^{14}\) Brealy, Myers, Allen (2005)
Graphs like these can been seen during the run, and are updated when new better systems are found. This is a nice feature which makes it easy to follow the evolution over time. The out of sample graph is a bit jerky which can be explained by the fact that there are quite few trades here at this point. This run went on for 45 minutes and final result is shown in figure 12.

![Training Equity Growth](image1.png) ![Out of Sample Equity Growth](image2.png)

**Figure 12: Result after 45 minutes.**

As can be seen in figure 12 the net profit in sample has improved from around 22000$ to 28000$ but out of sample has been slightly worse. Even though the out of sample graph may have some similarities with the in sample graph you want a much smoother curve that reminds more of the in sample curve. But for this example it will work. The out of sample curve might have had improved if the run were to be allowed to run for longer time, or it might have been worse. TSL have another nice feature where you can see a summary of interesting numerical data about the run. The performance of the out of sample data is shown in figure 13.
In this summary there are many interesting data for the run, like number of correct trades, standard deviation for the return and max drawdown any many other. In another window which is not shown here you can also see each trade made, the date of that trade the profit/loss of it and the cumulative profit after it. The next step is to choose which of the developed programs that will be translated to EasyLanguage code and tested in TradeStation. The most promising looking generated programs can be chosen by examining “run log graphs” and then match their fitness value with the saved programs fitness value. The system that is shown in figure 12 is the system that has best fitness value but that does not mean it is the most interesting system to look closer at. There might very well be system with slightly higher fitness error in sample but that looks much better out of sample and a system like that is much more interesting for the user. But for the example run I will use the system with lowest fitness error. There is a translator integrated in TSL that translates the generated program into EasyLanguage code. If you want to use the generated code in some other system there is options to save as java, C or Assembler code as well which gives the user many possibilities. The generated code is inserted into a strategy in TradeStation that controls how the trades are made. For this example I checked how well the developed model would perform from 20071101-20080701 and the result can be seen in figure 14.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Net Profit</td>
<td>6722.50</td>
</tr>
<tr>
<td>Buy and Hold Profit</td>
<td>3807.50</td>
</tr>
<tr>
<td>Avg%/yr(on 30D)</td>
<td>85.80</td>
</tr>
<tr>
<td>Gross Profit</td>
<td>14592.50</td>
</tr>
<tr>
<td>Total Number of Trades</td>
<td>27</td>
</tr>
<tr>
<td>Correct Long Trades</td>
<td>10</td>
</tr>
<tr>
<td>Incorrect Long Trades</td>
<td>4</td>
</tr>
<tr>
<td>Percent Profitable Long</td>
<td>71.43</td>
</tr>
<tr>
<td>Winning Trades</td>
<td>16</td>
</tr>
<tr>
<td>Largest Winning Trade</td>
<td>3030.00</td>
</tr>
<tr>
<td>Avg Winning Trade</td>
<td>912.93</td>
</tr>
<tr>
<td>Avg Win/Avg Loss</td>
<td>1.27</td>
</tr>
<tr>
<td>Max Drawdown</td>
<td>4397.50</td>
</tr>
<tr>
<td>Profit Factor</td>
<td>1.05</td>
</tr>
<tr>
<td>Std Dev Return</td>
<td>2.51</td>
</tr>
<tr>
<td>Total Flat Time, bars</td>
<td>3613.00</td>
</tr>
<tr>
<td>Sharpe Ratio</td>
<td>2.06</td>
</tr>
<tr>
<td>Percent Applied Data</td>
<td>20.00</td>
</tr>
<tr>
<td>Big Point Value</td>
<td>50.00</td>
</tr>
<tr>
<td>Buy and Hold Drawdown</td>
<td>8987.30</td>
</tr>
<tr>
<td># Eval Sys at Min err</td>
<td>28560</td>
</tr>
<tr>
<td>Fitness Error</td>
<td>0.000101357</td>
</tr>
<tr>
<td>Shrs/Ctls Traded</td>
<td>1.00</td>
</tr>
<tr>
<td>Trades per Year</td>
<td>45.45</td>
</tr>
</tbody>
</table>

Figure: 13 Summary for out of sample data.
As seen in figure 14 this model did not perform well but would have done a loss if it were to have traded live. This was expected because of the poor look of the out of sample graph is TSL. No performance measures should be drawn about TSL based on this example because this example was only made as a demonstration of the sequence of work for creating the model. A model with the performance out of sample as in this in example is usually not even worth to check in TradeStation but it still served as a good example of the process from start to a trading model.
4 MY WORK
This chapter presents the way this work has been carried out and to some degree how the work has been done over the time available for this final thesis.

4.1 COURSE OF ACTION
Before I started working with my final thesis I had never heard about genetic programming. This made the first time during my work mostly focused in gaining knowledge about it. During my work I have read three books about evolutionary programming and a big amount of articles covering the topic. One of the books was written by the creators of Discipulus which was good because they had many references in their book to Discipulus which was helpful. The directions I have chosen during my education, computer science and finance, suited this work very well. But the financial direction is theoretical and does not deal that much with actual trading. The focus rather lies on ways of valuating investments, companies, bonds, options and measuring risks etc. So some time in the beginning was spent on understanding the systems and how the trades actual took place.

4.1.1 Working with TSL
I started generating models in TSL quite early in my work even if I did not know that much about what actually was happening then and why. These early runs were useful for testing how the system worked and how it saved systems and so on. Before any meaningful work could be done I read the manuals to TSL and Discipulus which is about 250 pages together. As mentioned before TradeStation has its own programming language called EasyLanguage which also needed to be understood. This language is quite simple thou so that did not take that much time. When I felt that I had enough knowledge about how the system operated and the principles behind GP I was ready for starting some more structured testing. There is a huge amount of different instruments to test on so I had to choose if I wanted to do a few tests on many different instruments or if I wanted to do many tests with different settings on few instruments. I chose the latter and the instruments I have focused on are Standard & Poor’s 500 E-mini futures contract and the British Pound Future. The basic principle I followed was to test different fitness function with different entry, exit rules and if the system looked promising I tried them in TradeStation for another period of time which the models had no knowledge about. Later in my work I started to adjust the settings in the GP-engine to try to improve the results that way. I used a slippage and commission cost for a round turn of 20$ or 25$. In the beginning I did not use the more advanced features like stop loss, profit target etc. Out of sample used ranged between 20% - 30%.

Quite soon after I had started my testing a new version of TSL was provided. The main difference with this compared to the earlier was that it supported up to 60000 bars and the earlier only supported 30000 bars. It is more interesting to be able to use more data so I started to use the new version directly. After a while I noticed that the system behaved strangely when I used many bars. In TSL the training and the validation data is the same and in TSL there is one graph for the training and one for the validation. So because they are the
same the graphs obviously shall have the same appearance. When I was using more than 35000 bars I noticed that they were not the same anymore but the difference were quite small with 35000 bars but the more bars that was used the bigger the difference became. I mentioned this to the people at TSL and it turned out there was some large array bug in the program so I had to return to the previous version while this was being fixed. Another thing I noticed was that the Sharpe Ratio did not seem to be calculated correctly. Sharpe ratio is calculated like this:

\[
\text{Sharpe ratio} = \frac{(\text{Return} - \text{Riskfree return})}{\text{Standard deviation of returns}}
\]

During a particular run I had a Sharpe ratio of 0.60 on the training data and 0.61 on the out of sample data. But for all data, which is in sample and out of sample data combined, I had a Sharpe ratio of 3.28 which does not make any sense mathematically. I was in contact with them about this as well and got the answer that if less than 1 year of data was used the Sharpe ratio should probably be wrong. But I got similar results even when many years of data was used. I never got an answer why this was the case but it seemed it was some errors with the calculations for Sharpe ratio.

4.1.2 Working with Discipulus

When TSL is running there is not much you can do but wait until the run is over and check so the evolution proceeds and no unforeseen problem has occurred. During the first time this periods were good for learning other systems and read literature. But once I’ve learned the other systems and felt that I understood the theoretic aspects of genetic programming this time could be used more effectively. Fortunately I had access to a standalone version of Discipulus as well with which I could do some testing. An interesting case happened when I was just trying out Discipulus with some easy regression problems. With a polynomial of third degree like \(a^3 + b^2 - c\) Discipulus solved it fast. I expected the same to happen when I instead used a polynomial like \(a^5 + b^4 - c^3\). The unexpected event that occurred now was that Discipulus was not able to solve it all. This was kind of confusing but it turned out to likely be related to the generation of the data. For the first run I had generated the input to the data file in the way described by the pseudo code shown below.

```java
for(int i=-50;i<50;i++) {
    result=calculate(i+1, i+3, i*4-22);
    writeToFile(i+1, i+3, i*4-22, result)
}
```

So the input in the data file is in sequential order. On the other hand when I instead randomly generated the numbers that was written to the file Discipulus solved the problem in a reasonable timeframe. The time it took for Discipulus to solve it could range from 30 seconds up to 15 minutes. For the first example with sequential numbers it was allowed to compute during the whole night without being able to solve it. The only reasonable explanation I can see for what is causing this behavior should lie in the nature of the
residuals. The linear fitness function should be used if the residuals are distributed in a Laplace distribution and the square fitness function should be used if the residuals are distributed normally. When using the sequential approach the residuals are distributed in some other way. Another problem occurred when working with polynomial with high degree and squared fitness error. By their nature polynomials of high degree can be very big and when the error also is squared they get even bigger. In Discipulus the biggest error shown is equal to $2^{31}$ which is the biggest number the data type int can represent. The potential squared fitness errors when working with polynomials of high degree can be much bigger than that. When using the squared fitness function and having a pretty big range on the allowed inputs $x_i$, for example $x_i \in [-100,100]$, the performance shown in Discipulus never goes below $2^{31}$, at least not during the run I have done. The fitness calculation by itself does not use the data type int, which would have been very bad knowing the data types poor precision and the fact that it only represents integers, but rather it is of the type float. Still this behavior led me to believe there is some maximum value internally in Discipulus in some calculation which prohibits the evolution. If that is the case there is no way for the system to choose the better of the two individuals with very high fitness errors. Even if they both have very big errors one of them is still better than the other (if they are not exactly equal) and potentially very much better. But if there is a threshold internally individuals with fitness errors above this threshold will be regarded the same and there is no way for the GP to decide which one that might have potential for good further evolution and it might get stuck here when it cannot decide which individuals are better than the other. Obviously there is a restriction on how big the numbers can be that is restricted by how big numbers float can represent but that is such a big number that it is not a restrictive factor here. This is just a speculation from me and I have not been able to neither confirm nor dismiss it. When there is a constant involved in the expression looked for it can be hard for Discipulus to come up with an exact solution. This is the case because the constants used with standard settings are randomly generated and it is not certain that there is a combination of constants that combined in some way can take the value of the wanted constant. When this occur fitness errors very close to zero can only be found. Often a big amount of constants and trigonometric values of those is used in the final expression to get as close as possible in this scenarios.

I have also worked with writing my own fitness functions for Discipulus. The documentation available about this part was brief so it took quite a while before I understood how you had to code it before it actually worked. It was also written in C++ which I have limited experience from so that was also a problem although a minor one when the syntax of C++ much reminds of other languages like Java. When trying the generated models from the own fitness functions in TradeStation some problems occurred. TradeStation often generated error messages because the models made illegal operation like trying to take the square root of a negative number or a calculation overflowed. To be able to get around this problem some instruction had to be disabled in Discipulus. The problem that the numbers became too big for TradeStation to handle depends on the structure of the model. TSL have their
own rules for handling data in TradeStation so those problems do not occur when working with TSL and TradeStation. When a model generated in TSL is tested in TradeStation on the same time period on which it is built you want the system to give the exact same net profit, number of trades etc. This is very seldom the case thou and there is usually a small discrepancy between the systems. This differences it usually small so it does not really matter and is related to how TradeStation handles trades.

4.1.3 Advanced features in TSL
When we got the new working version of TSL with support for 60000 bars I started testing to build models based on the more advanced settings where the GP also will exit the market based upon certain stop loss levels or profit targets. The information about those are saved in higher level registers so to be able to use this functionality the GP system must be forced to use at least three FPU registers. The default setting in Discipulus is to use two registers and also Discipulus has a randomize feature integrated in the system that randomize the parameter settings between different runs trying to find the best setup. The randomizing over FPU registers used must be deactivated in this case. Normally only the first register f[0] is interesting where the result of the program is saved. But when using these other settings needed information is also saved in f[1] and f[2] and for other scenarios more registers may be needed. It turned out to be hard to get any good models when using these settings and with certain combination of settings no systems were generated at all.
5 RESULTS
This chapter presents the results of the evaluation of TSL. It is divided into three parts where the first deals with usability, the second with performance and the last with the models created by TSL.

5.1 USABILITY
To start a run in TSL if data is available is a very simple task and can be done by almost everyone. On the other hand if the user want to use more advanced features and understand why the system behaves like it does and how to adjust the system for creating better results good knowledge about how genetic programming works and in this case how Discipulus works is needed. Obviously the user need to have knowledge about the financial markets, different instruments and risk management but the potential buyers of this system can be assumed to have that because this isn’t a system a novice in trading should buy.

TSL comes with a manual with about 60 pages describing the program. Unfortunately this is not enough in my opinion because many things are not described good enough or not at all. Some of the things surely would have been easy to understand if one had been part of the actual development but for an outsider some things are hard to understand or not described thoroughly enough. On the positive side the author of the program has been very helpful and always answered questions fast with good descriptions. The problem that it is hard to understand some parts is also related to the fact that much of the code is hidden within DLL-files so it is not possible to see what actually happens there. With the knowledge of what is happening in those DLL-files much of the understanding problems would disappear. The problem with making the content available in the DLL-files is a copyright concern more than anything else because making those files “public” would give away how the system is created in some cases and it is understandable that the creators is not interested in that.

5.2 PERFORMANCE
The more data that is used the lower the evolution rate is for new models which is natural. When working with around 40000 bars the evolution rate usually is around 20 created systems per second on the CPU used during my work which was an Intel Core2 Extreme X9650. This CPU like most modern CPUs have more than 1 core, in this particular case the CPU has 4 cores. The problem with TSL, like Discipulus, is that it only uses one core. Discipulus have support for usage of more CPUs if separate modules are bought, one for each CPU.

GP is theoretically ideal for parallelization across multiple cores. The fitness calculations stand for the majority of the usage of the CPU in GP. The fitness calculations are independent in their nature and if these were to be calculated on different cores in parallel possibly huge efficiency gains can be made. As it is today TSL does not scale at all with multiple cores. Unfortunately the task of parallelize a single thread so it can be divided and run on different cores is not easy and very error prone. An even more interesting approach is to include GPUs and let the fitness calculations be done on them instead of the CPUs. GPUs
offer extremely high floating point calculation performance and a graphic card today typically has up to 128 different cores. Langdon and Bahnzaf have done an interesting project where they have used RapidMind\textsuperscript{15}, which is a commercial platform that parallelizes single-threaded applications over multiple cores, and a GPU for handling their fitness calculations\textsuperscript{16}. When much data is used speedups of 7-12 can be gain with the parallel approach relative the traditional.

TSL is a bit inefficient in the sense that it demands much attention from its user. First the input data needs to be prepared. After that all settings has to be adjusted and after the run the created systems has to be manually evaluated. If the user wants to do a new run he or she will have to redo the previous steps (except the data step if the same data is used). A run in TSL may take from a couple of minutes to a couple of hours depending of the amount of data that is used. This makes it very inefficient if the user wants to have a run during the night because only a fraction of that time will be used efficiently. What would be preferred is if there would have been a batch-like kind of feature integrated in the system where the user could set the parameters for many runs that would take place after each other. Only systems that lived up to a predetermined standard would be saved and when the user returned the next day or after the weekend the systems would be available for taking a closer look at. Ideally the input data should also be a parameter in the batch run and caught dynamically when needed. This is probably not as realistic when that data comes from another system but if input data were saved in different files it should be possible to switch data file between the different runs specified in the batch file.

During my work we have received new versions with improved performance which has been nice. Some of the versions was not enough tested before they were released thou and some of my work with the new version sometimes rather felt as beta testing because the systems behaved strange and I had to spent time looking for if I might have done anything different that might could have caused the problem. On the positive side it shall be said the when I have reported the problems they have been fixed very fast or I have gotten an explanation why they occur and how to come around the problem.

5.3 MODELS
This part will answer the most interesting question with this whole paper; do the created models produce good result out of sample? The short answer is unfortunately no. But the answer is not as simple as that. There is a big problem in the system that is related to how the systems are saved. During a run in TSL you can see how the result looks OOS and many times I have had very promising looking models just to find out that these systems weren’t saved when the run was terminated at a later point. As described earlier you can see graphs how good different models were during a run. When the run is terminated you can match a particular fitness error with a saved program and then use this promising looking model

\textsuperscript{15} http://www.rapidmind.net/
\textsuperscript{16} Langdon, Bahnzaf (2008)
evolved sometime during the run. The problem is that only certain systems are saved. This problem comes from how Discipulus saves system. Systems are only saved when a run is over and then a new run starts and new systems can be saved when this run is over and so on. In TSL the default termination criteria for a run is five generations without improvement. This means if a promising looking system is generated and then a better system is found there is no guarantee that the former system will be saved. Because the models are built strictly on the training data and takes no attention to OOS data a negligible improvement in training performance is considered better even if the OOS result gets much worse with that new model. If you were to decrease the termination criteria to one generation without improvement the chances that a particular system were saved would increase. On the other hand aborting a run after one generation without improvement is poor usage of the GP engine because the chances it will find better results later is big so then it might find no good solutions at all. If you instead were to increase generations without improvement to say 20 the chances for finding better models would increase but then the risk that the systems won’t be saved is also bigger. I would prefer if the systems that were saved was based on an average on the training fitness and the OOS fitness. The models would still be built only from the training data but the saved systems should take consideration to the OOS fitness. From a user perspective the only systems that is interesting is the ones that perform well both in sample and out of sample, and a system that is slightly better is sample but significantly worse OOS is of no interest. To sum up, the system might very well have come up with good systems the problem is I haven’t been able to test them because they weren’t saved.

Generally it has been hard to get any good models when much data has been used. The performance out of sample has not been good enough even thou in sample performance often have been good. This is a common sign for over fitting the training data. Most of the examples that are available have one thing in common; they are built upon quite little data. It is much easier to get better matches when you use little data, at least in sample. The reason for this is that it is easier for the GP to find a solution that matches fewer training cases than it is to find one for more fitness cases. Another parameter that has severe implications on the models is the usage of the round-turn cost for slippage and commission. If you set the cost for this parameter to zero you often can get systems that look very good, also when you test them in TradeStation for a period where they theoretically could have traded live. A system without trading costs will almost always make a huge amount of trades where the majority of the trades generate small profits. This is possible because there is no cost related to the trades. Unfortunately it is not possible to trade without costs. So when transaction cost are taken consideration to a system that make 1000 trades and generated and net profit of 10000$ without transactions cost the result will be a system that loses money. So building systems without transaction cost is basically pointless unless you are doing it for testing some new trading approach.

Some of the more advanced features integrated in TSL is not working very well because it seems TSL is built for always being in the market. It is hard to get the stop-loss and other exit
methods to work satisfactory. There is no point to cover a short position after a stop price has been reached and then moments after open a new short position in the same instrument.

As stated earlier I have had access to a standalone version of the engineering version of Discipulus. One thing I have worked with is to see how you can come around the fact that you cannot choose what system that will be saved. This is decided strictly by Discipulus and in TSL this is a big problem as described earlier. When you are creating your own fitness function you have direct access to the created programs via a pointer. So theoretically you can do whatever tests you want in the fitness function and if the program lives up to the criteria you have set up you can write the program to another file so you are sure that particular program actually is saved. The problem with this approach is that the generated program that the pointer points to is in the state of object code. This by itself is not really a program besides a bit tedious to translate it into a more understandable format. However the real problem is related to the fact that Discipulus is supposed to send a parameter to the fitness function stating the length of the program. Without this length parameter it is very hard, if not impossible, to know how big the program is and hence you cannot catch the program from memory because you do not know where it ends due to the fact of dynamic length of programs in GP. This parameter does not work as it should unfortunately and always gives the value zero leading to that catching the program is very hard. This makes it hard for TSL to have the opportunity to save systems when they want and this problem might have been something they have discussed during the development of TSL.
6 CONCLUSION

This chapter presents my impressions and my thoughts of TSL and automated trading.

6.1 FINAL DISCUSSION

It is hard for me to make any certain conclusions about TSL when the testing material is too small. What I can say is that I have not been able to produce any good models on Standard & Poor’s 500 E-mini contract or the British pound future which is the contracts I have been looking on. As explained earlier there is a possibility that good models have been created but these were not saved so I could not test them. There is also the possibility that if I would have looked on other contracts better models would have been created. The contracts that I have looked on are both highly-traded and therefore these contracts prices might be assumed to be efficient in the terms of the efficient market hypotheses. If a less traded contract was used possibly better models could be made there.

TSL is based upon a very interesting concept and if it can create models that can give risk adjusted excess returns it offers you a way to make money. A question that must be asked in this context is if you had created a system that could generate systems like that would you then sell the system?

It may very well be possible to use genetic programming to build models that give excess gains. I have no access to the source code in TSL so I have no idea if better written code there would have given better models. TSL may also be perfectly written but the problem rather lies within Discipulus. The only thing that is certain is that it will not be possible to get good models if there is no actual pattern in the data. If there is no pattern TSL and Discipulus can be perfectly written but still will not be able to create good models.

Automated models in general have a hard time performing during the circumstances on the markets the last months. The volatility on most markets is a lot higher than it usually is and movements of 5% in either direction have not been uncommon lately. Most models are not built for these circumstances and perform badly when they are used at the moment.

TSL is an interesting program but it has some important flaws at the moment. Much of the information about how the system works are hidden in DLL files which makes it hard to understand some steps in the way TSL works. With knowledge about these steps it would be easier to affect the process of generating the models. There are new versions with improved performance released at regular intervals so maybe one of these will solve the current problems.
7 REFERENCES

7.1 PRINTED SOURCES


TSL Manual.

Discipulus Manual.

7.2 ELECTRONIC SOURCES


TradeStation. <http://www.tradestation.com>
TSL. <http://www.tradingsystemlab.com>

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© Magnus Johansson