The Black Book of Financial Hacking

Third Edition

Developing Algorithmic Strategies for Forex, Options, Stocks

Johann Christian Lotter

ISBN: 1546515216
ISBN-13: 978-1546515210

Content

Preface

Iron rule for prefaces in trading books: Always mention your long trading experience in the first sentence. However, this might be a questionable recommendation. A veteran trader is normally ahead of a novice in one regard only. He has lost more money.

If you follow the price movement of a stock or currency on your PC screen, you'll soon believe to see patterns. The price drops a bit, drops further, then recovers. Surely it can't be so difficult to catch the right moments to buy and sell? However, when starting out in technical trading, you'll find yourself in a bizarre world, not unlike the medieval alchemist scene. Attempts to foretell the future - in this case, the price of a financial product in a few minutes or days - grow strange blossoms. Countless tools, called "technical indicators", are at the trader's disposal for the prognosis of prices and markets. And even less technical methods, such as astrology, candle patterns, Elliott Waves, or Gann Lines, are often and readily used - to the great joy of brokers.

When people want insight into hidden event chains, esotericism flourishes. And still the future remains unknown. Every year, private traders lose billions on the financial markets. It makes little difference how many trading courses they have attended or how many trading books they have read. At least, employed traders in investment banks or trading firms generate annual gains of about 3% above the market. Trading seems to work sometimes, albeit with rather unspectacular results. These are not achieved by experience or talent, but mainly by solid trade rules set by the respective company. No employed trader would attempt to predict tomorrow's prices with Elliott Waves or Gann Lines. Otherwise he would not remain employed very long.

Beginners have two major advantages. They are free from all the strange doctrines and the pseudo-knowledge that is circulating among traders and is uncritically regurgitated in trading books. And they know that they know nothing. So they have all the prerequisites to successfully hack the financial markets. Hacking is nothing illegal; it is simply a special way to approach things. Hackers despise gurus, experts and traditional "knowledge". They believe nothing and consider everything. They are always looking for new ways of solving old problems. They are practical and aim for quick results; theory suits them less. On the other hand, they have to be extremely competent in order to understand systems so deeply that they can control it. They rarely have a mathematical education, but expertly handle mathematical tools. This

way hackers are successful in many fields: computers, biotechnology, astronomy, cooking - and financial trading. Of course, they don't trade manually, but leave that to a self-written algorithm.

An algorithmic trading system is a software program that analyzes price curves of financial products and takes trade decisions autonomously. Start it, sit back and wait for the incoming profits: a great thing. In theory. If it works. Unfortunately, even a computer cannot predict the future. However, it can, unlike humans, indeed derive probabilities for the future price development from a price curve under certain conditions, similar to the weather forecast from a grid of global measuring points. Despite the limited prediction horizon, working algorithmic trading systems can achieve annual returns up to 100%. This sounds as it any capital could be multiplied in no time. But those returns are deceptive, since they come at the cost of an equally high risk. Even successful systems are only one nose ahead to chance. Mistakes in strategy development transform a theoretically profitable system immediately into a real loss system. The weather follows always the same laws and can be predicted in always the same way. But a trading system must not only be correctly developed and tested, but also permanently observed, adapted to the market and, if necessary, terminated in time. For this it is important to understand how and especially why such systems can work at all, and when they fail.

About this book

The book was written in parallel to my finance blog **http://financial-hacker.com**, which is dedicated to algorithmic trading. Other than the blog, the book is also suited for beginners with no previous knowledge in trading or programming. Since it was written from a hacker's perspective, it is practical. The example strategies are ready to be traded; the software and scripts can be downloaded from the blog.

The first two chapters introduce the basics of financial trading and programming. The following three chapters cover the development, optimization and testing of basic trading strategies with various methods for detecting trend and cycles in price curves. Since testing is the most important part of development, Monte Carlo analysis, walk forward analysis, and reality checks are also extensively discussed. The sixth chapter deals with machine learning for finding patterns in price curves. We'll then examine a system that, despite passing backtests with flying colors, would not work in live trading – and learn how to find that out beforehand. In the next two chapters we move on to long-term low-risk strategies, like algorithmic options selling and stock portfolio rotation. Finally, we'll build a scam robot – of course for educational

purposes only. The last chapter provides practical tips for developing your own systems and for achieving a regular income through algorithmic trading.

For advanced traders, the book offers insight in some black hacker tricks. Such as, faking phenomenal profits on a real, verified trading account - a method sometimes used by robot sellers. Or achieving better returns by intentionally triggering the dreaded margin call. And many more.

About the third edition

It's been 3 years since the first version of this book came out. The example strategies traded mainly currencies that are easily accessible to beginners. But since then we got many requests for long-term, low-risk systems. This edition has two new strategies of algorithmic options selling and stock portfolio rotation. Readers had also complained that the trend and counter trend strategies in chapters 3 and 4 were based on the same algorithms as the examples in the Zorro manual. I did not see this as a problem, but ok, they are now replaced with new algorithms. Which, as a side effect, also generate more profit.

I hope that this book may be useful for beginners as well as seasoned traders on their path to financial independence. Because successful private financial trading helps not only yourself - it helps the whole of humanity.

1 Dealing with money

Technical progress has enormously boosted productivity worldwide. It reduces the workload for producing goods and services annually by 2% to 5%, depending on the industry branch. Theoretically, we needed to work four to ten days less every year for producing the same goods in the same quality and earning the same income. Or we could work the same time and increase our income accordingly. Are we about to enter the paradise of prosperity?

Paradoxically, the opposite appears to happen. Working time increased and income – except for the very wealthy - decreased over the last decades in the industrial countries. US middle class families earned about $5000 less in 2015 than in 1999. Productivity growth seems to have a negative effect. Instead of working time, working costs are reduced; instead of wages the company profits grow. This has a short-term positive effect on the economy, but some unpleasant side effects in the long run - such as cyclic layoffs, unemployment, decline in demand and recurrent economic crises.

The remaining excess profits are invested. But not necessarily in research, development, or production of goods. More and better products would often not meet a shrinking demand. The profit surplus is instead invested into the financial markets. Here's always demand. Hedge funds, banks and trading firms are pumping huge chunks of money through the global markets. Every single day, about 7 trillion dollars in financial products and currencies change hands. To compare: About 270 billion dollars are spent per day for wages, about 210 billion for goods and services. Speculation in securities, currencies, and derivatives produces an enormous shadow economy that exceeds the real economic output many times, without however producing any useful things. But it generates interesting economic fluctuations with occasional bank and market crashes.

Just as the financial markets dwarf the real markets, so the speculative gains exceed by far the profits from entrepreneurial activity. Ideas and achievements are outdated - today the largest profit factor is money itself. It thus accumulates increasingly in the hands of those who possess it already. Thomas Piketty has coined for this the term *patrimonial capitalism* - capitalism based on inherited wealth. The move from dishwasher to millionaire is more

difficult, the move from millionaire to billionaire easier than ever. Moderate inequality can promote the effectiveness of an economic system. But in excess, and with no relation to personal skills and performance, inequality destroys in the long run the social consensus.

There are ways to counteract this tendency. One of them is to beat the financial markets with their own weapons. By participation of many private traders, even and especially from developing countries, a good portion of the circulating money could be brought back into the normal economic cycle. This would increase the global demand and boost the global economy. At the same time it would dampen the profits of large funds and financial institutions, and make investments into real products more attractive. The private trading is therefore important and useful to society, even though it produces nothing, but only redistributes money. Actually, it should be rewarded by the government, for instance with a high tax allowance on trading profits. However, a precondition for the benefit is that these profits are really achieved. And that is currently not the case. In the long run, almost all private traders lose money. So trading has the reverse effect that it is supposed to have. The money vaporizes into the financial markets and is lost for the real economy, this way further increasing global inequality.

It is therefore not only in the personal, but also in the general interest to trade profitably. For this one must first understand the system and methods. How does financial trading work at all?

Trading 101

The image most people still have is the 'pit' – the stock exchange floor where hundreds of traders are shouting and gesturing to one another. Those guys are endangered by unemployment, since most trades are today handled electronically. The electronic markets use computer networks to match buyers and sellers. While this system lacks the romance of the stock exchange floor, it's efficient and fast - and anyone can participate in it.

For this you need a computer and Internet access, and a **Broker** to handle your trades. Your broker connects you to the networks, places your orders, and even lends you money for trading. Most brokers offer **Demo Accounts** (also called Paper Accounts or Game Accounts) where trading can be practiced without risking real money.

The prices of financial products are constantly changing, often within milliseconds. The point of trading is making a profit from the difference between the buying price and the selling price. **Day Trading** is the buying and selling of financial assets in short time intervals. There are several different styles of day trading, ranging from extreme short-term trading such as **Scalping** where positions are only held for a few minutes, to longer term **Swing Trading** where a position may be held for many days or even weeks. If you keep stocks for years, you're not a trader but an **Investor**. And if you invested a lot of money in a PC that's directly connected to the stock exchange, you can do **High Frequency Trading** and buy or sell in microseconds.

All those trade styles not only differ in trade duration, they require completely different trading methods and algorithms. And one of them does normally not work at all. Due to transaction costs, scalping cannot be profitable (except for brokers and system vendors). No working scalping system for private traders is known - otherwise it would be nice to see one's account grow by the minute.

If you have your TV running 24/7 on a news channel, read 12 newspapers per day and base your trade decisions on news from CNN or the Wall Street Journal, you're a **Fundamental Trader**. If your trade decisions are based on price curves, you're a **Technical Trader**. Technical trading does not require that you know anything about the asset that you buy or sell; you don't care if it's pork bellies or the Euro, you're only interested in its price curve.

A financial **Asset** - also sometimes called **Instrument** or **Symbol** - is the object you trade with, such as a currency, commodity, stock, future, option, ETF, CFD, or another derivative. In currency trading (or 'forex trading', **Forex** = Foreign Exchange) you buy or sell one currency and pay in another currency. For example, when you buy the asset "EUR/USD", you buy in fact the Euro and pay in US Dollar, and when you sell "USD/JPY" you sell the US Dollar and get paid in Japanese Yen. You always trade the first currency against the second currency. It gets funny when you're European and buy EUR/USD, which means you buy EUR and are charged in US Dollar for which you pay with EUR.

A special derivative, available from brokers that are mostly registered in Cyprus (or not at all), has become popular among beginners lately. The **Binary Option** - not to be confused with real options that are covered in another chapter of this book - is a bet that the price of an asset will rise above (or fall below) the current price after a certain time period. If you are correct, you won't win the full stake, but only about 75% to 95%. This way the bookie, pardon, the broker is sure that you're losing in the long run. Compensating

an average loss of about 10% per trade is not easy. On my blog I've described a method to still win with binary options, but it's a bit messy, and even if you succeed it's not sure that you get hold of your winnings. Payout requests often meet unexpected difficulties with binary brokers. Binary options have only one advantage: Since they are regarded as gambling, their profits are tax free are in some countries - which in practice, however, has not much significance.

A more serious derivative is the **CFD** (**C**ontract **f**or **D**ifference). It is especially popular in Europe. A CFD is also a sort of bet on a rising or falling price. Many brokers don't offer real stocks or commodities, they offer CFDs instead. When you buy a CFD, you are not buying the underlying asset, although the movement of the CFD is directly linked to the asset price. You just have the right to pocket the gain (if any), and have to cough up the loss (if any). Since you do not own the asset, you do not need to pay for it - instead you place a deposit with your broker, called **Margin**.

The margin deposit is for covering a possible loss of your trades, since the broker doesn't want to have to go after you for collecting his money. The margin can be as low as 0.5% of the real price of the asset. This means you can trade a volume of up to 200 times your capital. This factor of 200 is called **Leverage**. It is a convenient way to realize huge profits, or to lose all money in no time.

The **Balance** is the current money in your broker account – your initial capital plus all wins minus all losses. The **Equity** is the money you had if you closed all currently open trades. Due to leverage, the value of trades can be negative, thus the equity can be less than the balance. The equity changes all the time together with the prices of your assets, while the balance only changes when you buy or sell something. When the trades don't go well and your equity drops below your total open margin - the deposit for open trades, see above - your broker will start closing your trades, like it or not. This is the notorious **Margin Call**. It does not necessarily mean loss of all capital: Since the trades are closed, their margin is freed and remains on your account. We'll see in a later chapter how to even achieve profit with margin calls. Still, with normal trading systems better avoid that situation.

Contrary to popular opinion, high leverage does not automatically mean high risk. Quite the opposite, with high leverage you need less margin for the same trade volume and are thus safer from the dreaded margin call. The higher the leverage, the safer you are - but the less remains on your account if it still

happens. In extreme cases – when the broker can't close your trades immediately for lack of market liquidity - your balance may even become negative. So the broker will demand additional money for closing your account.

The **Ask Price** of an asset is the price at which you can buy; the **Bid Price** is the price at which you can sell. The Ask price is normally higher than the Bid price. The difference between Ask and Bid is the **Spread**: Ask = Bid + Spread. For currencies, the spread is usually in the range of 0.01..0.05 percent of the price (or about 1..5 pips, see below). That's a lot less than the ~2% that you usually get charged by banks or exchange offices! If you buy an asset and sell it immediately, you lose the spread. In addition, brokers often charge a **Commission** for buying or selling an asset. For brokers that waive commission, the spread is correspondingly higher.

When you have a **bullish** view, meaning you think the price will move upwards, you **Buy Long** (or just 'buy'). That means that you buy a certain amount of the asset at the ask price, and have to sell it later at a higher bid price for making a profit. The price must rise by at least the spread for making profit at all.

When you have a **bearish** view, meaning you expect the price going down, you **Buy Short** (or - somehow confusing - 'Sell Short', or just 'short'). That means you sell an asset at its current bid price without owning it. Therefore, you have to **Cover** it - i.e. buy it back at its ask price - at some point later. The asset price should have fallen in between by at least the spread for making profit. Buying short is a bit counterintuitive, but is in fact the same procedure as buying long, only with the ask and bid prices reversed. Many human traders prefer to buy long rather than short, for psychological reasons or because the broker has a restriction on shorting. This is visible in the charts, especially with stocks - the price movements are not symmetrical in time, but show a sort of sawtooth pattern. You can exploit this in strategies.

Entering or opening a long/short position, and **Exiting** or closing a position are just other words for buying and selling an asset.

A **PIP** (**P**oint **I**n **P**ercentage) is a certain fraction of the price of an asset. Prices normally only move a little tiny bit during a day, so it's more convenient to say "The EUR/USD is down 15 pips" than "it's down 0.0015 Dollars". One pip is a unit of the last digit of the price. In most currencies one pip is 0.0001 in units of the second currency. One exception is USD/JPY, which has only 2 digits after the decimal - 123.45 - and thus one USD/JPY pip is equal to 0.01 Japanese Yen. There are different pip values for stock, index, or commodity CFDs. Many brokers display **PIP Costs** for their assets. That

is not the price of one pip, but your profit or loss when you buy one lot (see below) of that asset and the price moves up or down by one pip.

A **Lot** of an asset is the smallest number of contracts you can buy. Few brokers allow you to buy currencies in multiples of 1 unit; most offer 'micro lots' of 1000 or 'mini lots' of 10000 currency units. So you can buy 10000, 20000, or 30000 EUR/USD contracts, but not 1, 100, or 4711. For CFDs the lots are different, but they are often in the range that one pip loss or gain is about 1 dollar. The margin required for buying one mini lot is in the $50...$100 range. The lot/pip system is an abstraction layer that puts all prices, losses, gains and margins in about the same range, and makes assets easily exchangeable.

A **Trade** consists of at least two instructions (**Orders**) to the broker, a buy and a sell, which are transferred electronically. You can either buy at the current asset price (**Market Order**) or specify an **Entry** price that automatically triggers the purchase when reached within a certain time (**Pending Order**). When buying you can already place a pending sell order, which shall be executed when the price moves above or below a certain limit. A **Stop Loss** or simply "stop" is a price limit in loss direction, at which the asset is automatically sold for avoiding further losses. A **Take Profit** (or "profit target") is a price limit in favorable direction, at which the asset will be sold automatically in order to bag the win. With a stop very distant from the current price, and a very close profit target you achieve a high **Accuracy** (or "win rate"), meaning that you will win a lot more trades than you lose. Why this still brings no wealth we'll see below.

It annoys traders especially when an already winning trade falls back into the losing zone by a reversal of the price trend. To prevent that, you can tighten the stop loss automatically (called **Trailing**) when the price moves in favorable direction, on and even above the original purchase price.

Orders require a certain time for execution; so the price at which they are **filled** can differ from the price that triggered them. This change is called **Slippage** and affects profits and losses - mostly to the trader's disadvantage. Since one cannot predict the slippage, many traders suspect their brokers to manipulate it. Which brings us to the next topic.

„There's gold in them hills!"

Private financial trading evolved under the name of "day trading" into the modern day equivalent of a 19th century gold rush - with the same participants today that you had back then. At the low end of the food chain are the gold diggers - the private traders who stare at price curves all day, waiting for the right moment to buy or sell. They are inspired by tales of lucky guys who found huge nuggets and became fabulously rich. Next in the food chain are the suppliers - the sellers of systems, books, trading seminars, consulting services, or trading robots. They prey on the traders and make real money this way. Many of those suppliers are ex-traders who realized where the gold really lies. And at the top of the chain are the large mining companies: hedge funds, investment banks, and other trading institutions that employ professional mining methods, such as algorithmic trading and high-frequency trading. Not to forget the broker firms that earn good money by letting a large number of private traders dig on their territory.

A side effect of any gold rush is lawlessness. Everybody cheats everyone. Nothing is as it seems. It's not only about money; it's also about prestige among traders. If all reports on trader forums about fantastic trading methods and profits would be true, trading would be the most simple and profitable job in the world. Reality is different. According to broker statistics, their customers lose on average 13 pips[1] per trade. Significantly more than a loss caused by opening and closing positions at random. Consequently, private traders would be well advised to use dices for their trade decisions. It's said in trader circles that 95% of all beginners give up within the first twelve months after having lost most of their capital. For the 95%, however, I couldn't find any evidence or source. The number is probably exaggerated, since it would need a lot of effort to mess up so many trades. According to brokers, 'only' 80% of their customers lose money in their first year of trading. For all following years the loss quote is about 65% - for beginners as wells as for trading veterans. There seems to be no relation between trading experience and success. Not even with professional traders, as demonstrated

[1] For EUR/USD; with other assets the losses are similar. Source: FXCM statistics for 2009-2010, published on DailyFX.

in studies by Daniel Kahneman and others. Still, 35% of private traders finish their financial year with a pleasant profit. Unfortunately they lose it in the next year in most cases, since the 65/35 quote still applies. But from this quote we can also conclude that some private trades must belong to the lucky 35% for many years in a row[1] - simply for statistical reasons. These traders become famous, write books and brag with their successes on trader forums.

Not so lucky traders often blame their losses on the evil broker. Look at that trade that again just went belly-up: Right after hitting the stop loss, the price jumped back and would have made me rich if the trade were still open. No doubt, the broker has closely observed the trade, and hit the button at the right moment to stop it out early. This practice is called "Stop Hunting". Especially in the US, traders tend to suspect their brokers of this or similar schemes and sue them after having lost their money. This is one of the reasons for the bizarre NFA and NASD rules (NFA = National Futures Association; NASD = National Association of Securities Dealers). Brokers are, for instance, not allowed to accept stop-loss or take-profit orders from US citizens. Which are also not permitted to open more than four positions per week as long as they're not rich enough. Those rules do not benefit anybody, but irritate developers of trading software for the US market who have to program around them.

Of course, brokers are not busy observing their client's trades. And normally they even don't want them to lose; the more the clients win, the more they trade, and that is good for the broker. His income is spread and commission, not lost trades. But there is a big IF. If a broker is also a "Market Maker" - meaning that he creates his own financial products (f.i. binary options or CFDs), calculates their prices, and deals directly with the client instead of transferring the trades to the global markets - every loss of the client is a profit for the broker. The trader does not play against the market, but against the broker. Some brokers, it is said, improve their income by letting their customers lose a bit more than the 13 pips per trade mentioned above. Such

[1] Assumed there are 5 million of private traders worldwide, approximately 25.000 should have made consistent profits in the last 5 years. Of course, without being to blame for that.

dishonest brokers are called "Bucket Shops". Of course they don't manually manipulate their client's trades, but use a special software. The "Virtual Dealer Plugin" of a certain popular trading platform allows brokers to generate artificial slippage and apply other methods for maximizing their profits at their client's cost. How many brokers actually do that is not known. Even though almost every forex broker uses the mentioned software, most of them (at least I hope so) don't activate this secret "Feature".

The real scam happens elsewhere. Most private traders realize sooner or later that they are losing money, no matter if their broker cheats or not. Apparently their trading is not good enough. Fortunately there's help. It's offered by countless trading schools, consultants, book authors, and vendors of trading robots and colorful technical indicators. You can easily spend a five-digit amount for a multi-semester trading course by a famous instructor. And you have to hurry, since the last course will be sold out soon due to high demand. For what can money be better spent than for becoming a top trader? Answer: for almost anything.

Manual trading with no inside knowledge is a rather simple activity. Once the basics are understood, more training won't make anyone a better trader, just as more roulette playing won't make anyone a better roulette player. Trading success is mostly uncorrelated to experience, to intelligence[1], or to a 'trading talent', whatever that might be. In the mentioned studies by Kahneman no employed trader could achieve significantly higher long-term profit than any of his colleagues, no matter with which method she traded[2]. Nevertheless, books[3], courses and workshops teach lots of new trading methods with lots of fanciful names. Usually they are offered without any justification, explanation or theoretical basis – just as if the praised "indicators" were a kind of magical oracle that only needs to be interpreted the right way. The entry and

[1] The physicist Isaac Newton traded around 1720 with South Sea Company shares. After the loss of 20,000 pound sterling (about 6 million dollars of today's gold value) he finished his trading experiment with the comment: "One can calculate the orbits of the stars, but not the madness of men".

[2] Consequently, Kahneman advised the trading firms to abandon the bonus system. The advice was not heeded.

[3] Of course there are also reasonable trading books. This one for instance.

exit recipes have to be followed to the point, like the spells in ancient conjuring books. They are always accompanied by charts with great profits. And they make you feel that you now know the secrets of trading. But strangely, they don't make you a top trader. Quite the contrary, after a while you'll notice that you're now losing money faster than before. And this despite all the good-looking charts, all the enthusiastic reviews on the website, and all the other traders that became fabulously rich using that method.

According to CFTC (**C**ommodity **F**utures **T**rading Commission) statistics, the average victim spends about 15,000 dollars during her trader's career for trading systems, guru seminars, magical indicators, forex autopilots and similar offers. And why should such a lucrative business not attract many vendors? After all, they do their very best to deliver a product that does not really exist, but is nonetheless in extreme demand. Instead of a real trading secret, they have to come up with something that appears so. As long as people are willing to pay, other people will gratefully take the money. It's not the vendors, it's the greed and gullibility of the buyers that produces all the junk.

For the sake of completeness, one must mention that a few of those top secret systems really worked for a while. The most famous example was the "Turtle Trading System" of the 1980s. It was one of the first algorithmic systems, although executed by human traders, not by computers. Backtests show that it indeed produced profits for almost 10 years. Unfortunately, the markets have changed. But even today, some relatively simple algorithmic systems can make relatively constant profits in good years. Let's examine the requirements for that.

Ticks, bars, candles

An algorithmic trading system, also called **strategy**, is a program that runs on your PC and trades for you. Therefore it generates a passive income. The system behaves like a technical trader. Buying and selling decisions are usually based on the price curve of the respective assets. Sometimes additional data are evaluated, for example the market volume, the prices of related assets or other data sources such as the "Commitment of Traders" report. In his novel "The Fear Index", Robert Harris described a trade algorithm that detects fear by evaluating mass media, and uses it as indicator. Principally it would not be very difficult to realize such an algorithm.

A **price curve** is an abstraction. Firstly, there are two for each asset, one for the ask and one for the bid price. But even these are not real curves - they consist of many separate price quotes by the market participants. This results

in a point cloud rather than a curve, where every point – also called **tick** - is a buy or sell offer at a certain price and time. Adjacent ticks can be quite different, both in price (by a few tenths of pips) and in time (by a few tenths of a second).

Since point clouds are difficult to handle, the ticks of a price curve are traditionally combined into **bars**. These are not the pleasant places where alcoholic liquids are traded, but periods of time. A bar may contain all ticks of a minute, an hour, or a whole day, depending on how you want to look at the curve. The **chart** in Fig.1 shows such a price curve with 1-day bars:

Fig. 1 - EUR/USD 2009, Candlestick Chart

This was the EUR/USD exchange rate in the summer of 2009. The vertical axis is the price, the horizontal axis the date. Each bar is represented as bright or dark **candle**, which is why this view is called a **candlestick chart**. The horizontal width of the candle is the bar duration (here one day), and its height is the price movement during this period. In bright candles, the price rose up; in dark candles he fell down. The fat **body** of a candle reaches vertically from the initial price (**Open**) - the first tick - until the final price (**Close**), the last tick inside the bar. The thin **wicks** above and below the candle represent the highest (**High**) and the lowest tick (**Low**) during the bar period. Example: If you look at the candle from July 1 (on the vertical line that is marked "Jul"), you see that the day began with an open tick of about

1.4100. The highest tick of the day was approximately at 1.4150, and the lowest at 1.4000. The day ended with the close tick at 1.4050. Since this value is lower than the open tick, the candle is black.

If you count the candles of each month, you will find that there are less than 30 or 31. This is because trading is suspended at the weekend. For traders, the week begins on Sunday 22:00 GMT when it's 8 AM in Sydney, and ends Friday 22:00 GMT when it's 4 PM in New York. On weekends, there are no ticks and thus no bars and candles. On the horizontal time axis of a chart the weekends and holidays are simply missing.

In past times, a bar was really equal to a day. The closing price was the price of the asset at 4 PM when the market closed, and the opening price was the first quote next morning at 9:30 after the traders had contemplated their strategies all night. So the width of a candle was 6 hours and 30 minutes, followed by a gap of 17 hours and 30 minutes during which no price quotes arrived and no trading took place. In modern times, many assets are traded online 24 hours a day, so there is no real opening and closing. There are no gaps (except for the weekend), only sessions when more or less people are trading. The trading session starts on Monday morning in one part of the world, and ends on Friday afternoon in another part of the world. That covers about 5 days per week. Because the big stock exchange places are in New York, London, and Tokyo, the days are broken up into the U.S. session, the European session, and the Asian session. Each session is roughly from 9am to 5pm on the local time. These sessions do overlap a little, and at those overlap times you have the most people trading.

When an asset is traded around the clock, as in the above chart, the closing price of one bar is usually almost identical with the opening price of the next bar. The appearance of those candles has not much significance for trading, since it depends on the time zone of the viewer. Day bars from midnight to midnight contain other ticks as bars from noon to noon and therefore produce with the same asset on the same day a completely different candle. Only assets that are traded at a certain exchange still have fixed appearance of a day candle and a gap from the close to the open of the next candle. For forex, commodities and most CFDs, however, a bar is just an artificial division of time without deeper meaning.

The price curve properties that can be exploited for algorithmic trading depend not only on the asset, but also on the bar period. Although price curves look **fractal** - you normally can't tell apart a 1-minute-chart and a chart with 1-day candles - their statistical properties often vary considerably, since with

other time scales other market participants come into play. Thus by simply changing the bar period you can use very different strategies for the same asset. This also means that a trading system normally works only with the asset and bar period for which it was designed. Systems working anytime and anywhere would be nice, but no one has yet discovered them.

Some traders believe that bars covering not a fixed time period, but a fixed price period, give them a better insight into the market. With a focus on price movement, long periods of consolidation are condensed into just a few bars, thus highlighting "real" price trends. There are many special bar types: Renko Bars, Range Bars, Momentum Bars, Point-and-Figure Bars, or Haiken Ashi Bars. The **Renko Bars** in Fig. 2 have a fixed price range; a new bar begins when the price has moved more than 1 cent. All those special bars make the price curve look somewhat smooth and more predictable. But this is mostly an illusion, as it's achieved by losing relevant information, such as the speed of the price movement. Therefore, price-range or other special bar types make the price curve harder to predict, not easier, and do not make much sense for algorithmic trading systems.

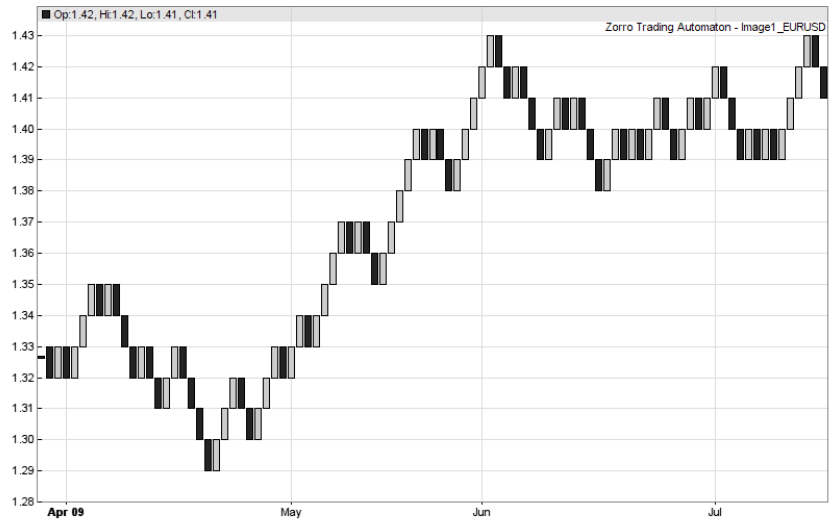


Fig. 2 – EUR/USD, Chart with Renko Bars

Frequently used bar periods for algorithmic trading are 60, 240, or 1440 minutes. Technical traders sometimes use a price curve of 1-minute bars or even of single ticks for **scalping** - buying and selling in very short time intervals. It is indeed tempting to make profits within minutes. And as an additional bonus you're getting a lot more bars and trades when backtesting a

scalping strategy. That's a great advantage, because the quality of test and optimization increases with the number of trades. Timely price data is always in short supply. Still, scalpers tend to lose their money much faster than other traders, and this for good reason. The shorter the trades, the smaller their possible gains, and the larger in relation are transactions costs such as commission, slippage, and spread. You can see in the following diagram what this means:

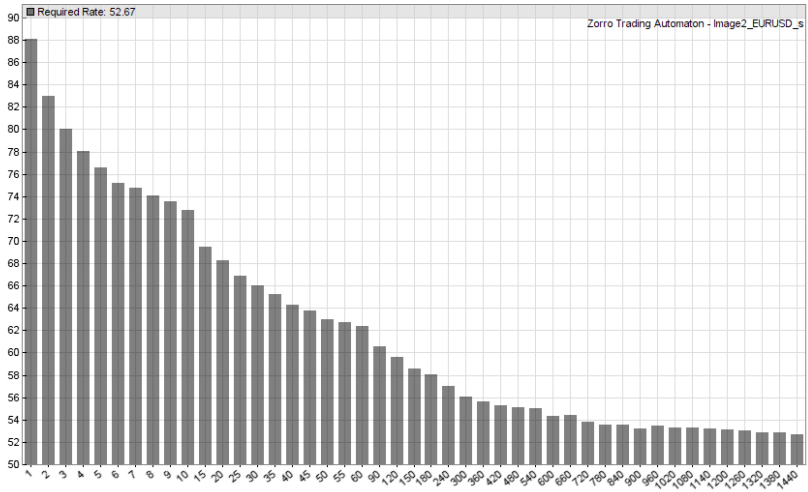


Fig. 3 - Required win rate vs. bar period (EUR/USD)

The vertical axis is the required profit rate in percent for breaking even with a trade duration plotted on the horizontal axis bar in minutes. It is assumed here that a trade is opened at the beginning of each bar and closed at the end. The gain or loss per trade thus is simply the height of the bar's candle, without the wicks. As can be seen, one would have to win 88% of 1-minute trades to

cover the cost by spread and commission[1]. But with one-day trades a 53% win rate is sufficient. For this and other reasons, algorithmic trading systems operate normally with bar periods in the range of hours and days.

The exception, of course, is **high-frequency trading**. Those systems use no bars and don't predict prices, but derive profit from the price differences of individual ticks. This requires some prerequisites that are, unfortunately, normally not available to the private trader. For example a direct and very fast access to the market and to the quotes from all vendors. And virtually no transaction costs, since minimal price differences are exploited. It won't work with a normal broker. Private traders must therefore rely on other methods for trading algorithmically.

How does a trading system work?

The price curve in Fig. 1 looks rather chaotic at first glance. One can identify a rising range - a **trend** - from mid-April to the beginning of June. If one had bought at the beginning of the rise and sold at the end, one would have made a nice profit. But how do you know beforehand when a trend will start, and when it will end? The curve does not seem to help, since within the trend and in other parts it shows many irregular fluctuations. It is obviously subject to many different influences besides the trend. These influences come from the **market**, the sum of all buyers and sellers of the asset. In the case of the EUR/USD rate, this would be all banks and traders that exchange euros for dollars, or vice versa.

In a perfect, efficient market, prices would only be affected by real events. For company shares this would be the publication of quarterly results or the announcement of a new product. For currencies it would be a change in the central bank policy or the publication of economic data. In such a perfect

[1] Assumed is a bid-ask spread of 0.3 pip and 60 cents commission per 10000 contracts, as usual for forex. The diagram has been created with the script **Image3.c** that is – as the other scripts for all diagrams in this book. – available on the Financial Hacker website for experimenting with other parameters and assets.

market there are only fundamental traders who have all the information, decide always rationally and act without delay. They react to public events, but also to their own needs. If company results are good, the value of the stock increases, which is reflected in a jump on the new price. When a trader needs cash, she sells her shares. Or she exchanges foreign currency to secure an export business. Public events produce sudden price jumps, private needs cause smaller irregular price movements, a "noise" on the price curve. Since private needs or public events can normally not be foreseen - at least not without insider knowledge - the price curves of such an efficient market are pure 'random walk' curves with no information for the prediction of future prices.

Fortunately for trading strategies, real markets are far from this theoretical ideal. Market inefficiencies are everywhere, sometimes less, sometimes more. They are caused by slow reaction on news, incomplete information, rumors, irrational trading behavior (such as following advices by trading gurus) and predictive behavior (when traders react on the price itself). Any inefficiency causes a price curve **anomaly**, a deviation from randomness. It can allow a strategy to predict the near future of a price curve with some degree of accuracy. Of course the question is now: How can you detect anomalies in price curves?

Look at the two curves below (Fig. 4):

Fig. 4 - Price curve and random walk

One of the two lines in the above chart is a real currency price in US$, the other one is a meaningless curve from summing up random numbers. Can you tell which curve has real prices? If not, don't worry: In studies, even 'expert traders' and analysts were unable to distinguish between real prices and

random numbers, let alone to identify the asset or to predict a future part of the curve. But the similarity of the two curves is superficial. Prices don't walk randomly. They are not produced by a random number generator and have many interesting properties that don't appear in random walk curves.

But this is a task for a computer. Unlike humans, computers have no problem at all to identify the real curve[1] in Fig. 4. For this they can use many methods. One of them is the distribution of price movements:

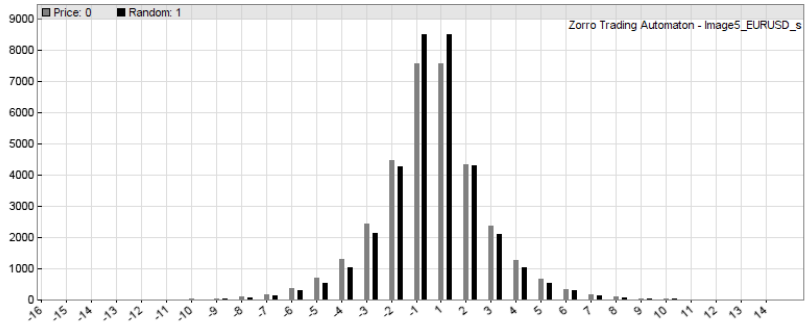


Fig. 5 EUR/USD, frequency vs. duration of price moves

In the chart, the height of the bars is equivalent to the number of rising or falling price series. The grey bars are from the real EUR/USD price curve above, the black bars from a curve of random numbers. The numbers on the x axis are the duration of a price movement in hours, at the right side for rising, and at the left side for falling prices. The height of a bar indicates how frequently such a rising or falling series occurs in the curve. For instance, the bar above the **3** at the right shows how often the price was rising for 3 hours in sequence. If prices would move totally random, the grey bars had the same heights as the black bars. We can see that this is not the case: rising/falling series of 3, 4, 5, or more hours occur more often in the real price curve than in the random data. 1-hour series - a price rising in the first and falling in the

[1] The bright curve is the EUR/USD price.

second hour, or vice versa - occur less often. Real prices tend to keep their direction. This effect causes the famous "fat tails" of price distributions. It exists with most assets and time frames, like minutes or days instead of hours.

One could think that this effect is easily exploited to the trader's financial advantage. Simply wait until the price went up during two bars, then buy. With a probability slightly above randomness – about 52%, dependent on the observed price curve – the price will also rise during the third bar, allowing us to sell at a profit. Is this already the path to quick riches? Unfortunately not – even if we could eliminate all trading costs. The markets deny too easy profits. The 48% losing trades lose a bit more than the 52% winning trades, this way equalizing profit and loss. You can later test this with simple scripts and different assets. By the way, this effect is the reason why binary option brokers always offer worse odds than 1:1.

But there are other inefficiencies in price curves that can be really exploited in trading systems. One of them is visible in the following histogram (Fig. 6):

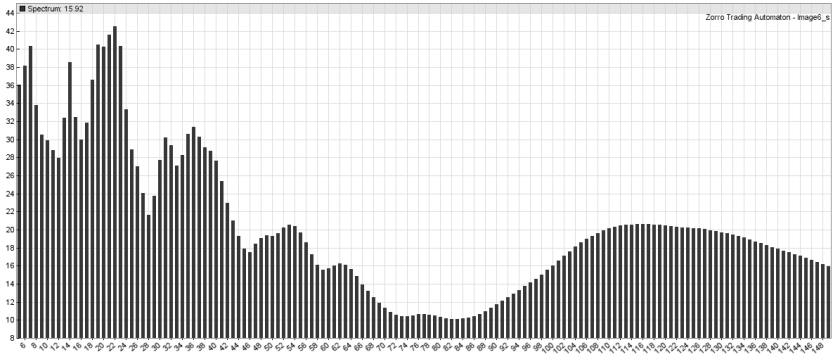


Fig. 6 S&P500, frequency spectrum

This is a **spectral analysis** applied to the S&P500 price curve in the last week of January 2013 (S&P500 = Standard & Poor index, combined value of the 500 main US stocks). A spektral analysis detects regular **cycles** in price curves. Cycles are caused by several effects, for instance when traders close profitable trades at a different time than losing trades. This can synchronize the behavior of a large number of traders and make the price curve swing up and down. Other cycles can be by daily or weekly trade behavior patterns. The price curve is therefore often a superposition of cycles with various lengths – a typical inefficiency not appearing in random curves. Those cycles are visible in the „peaks" of the displayed spectrum. The x axis is the cycle duration in hours, the y axis is the amplitude, the strength of the cycle. One

cycle has about 24 bars, equivalent to a trading day. You can identify other cycles in the area of 40 and 60 bars.

Most such cycles are shortlived, but if a cycle is stable for days or even weeks, a trading system can use it for profitable trades. The following chart (Fig. 7) shows a different view on the situation. We can see the same price curve from the spectral analysis. Its cycle of 24 bars was filtered out of the curve with a bandpass filter, and plotted below the chart:

Fig. 7 S&P500, price curve and 24 hours cycle

You can see that this filtered cycle follows the peaks and valleys of the price curve. Sometimes it even runs ahead a bit. Buying at the bottom of a valley and selling at the top of a peak would be a rather simple system, but would – as you can see by applying those trades to the price curve above – produce some profit. At least as long as the cycle holds steady. The fact that the S&P500 price rised during the month does not matter for the system. In one of the following chapters we'll develop a system that trades with such a filtered cycle curve.

The market often produces temporary price movement patterns that can be used by intelligent algorithms for forecasting short term trends. The following curve was generated by a neural network with a "deep learning" algorithm (Fig. 8):

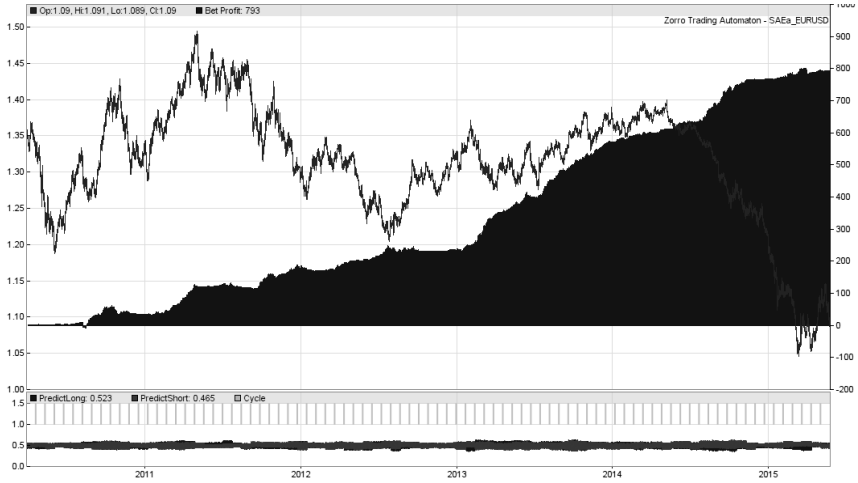


Fig. 8 - Neural network, hit curve

In a neural network the data packets pass several layers of ‚neurons' that process them and derive a prediction. From the outside it's a Black Box. Its decision criteria are opaque. The neurons wire themselves in a training process that minimizes the prediction errors. This way the network learns by trial and error. Certain training algorithms allow the network to detect patterns even in extremely noisy data, such as price curves. The above network uses such an algorithm. It analyzes the price differences of the last hours and predicts the EUR/USD price trend of the next 60 minutes. The black area is the sum of all correct predictions minus the wrong predictions. On average the network is right in 56% of all cases[1], which is a lot better than the 52% with the above mentioned simple 2-bars trend. In the test the network was re-trained every 4 weeks, since price patterns are short-lived and the hit rate begins to

[1] In publications you can find price curve prediction rates of 70% or even 80%, allegedly achieved with neural networks. Therefore the involved scientists should be multi-millionaires meanwhile. Problem is that you can easily produce any arbitrary prediction rate in hindsight, by creative selection of the test period and the network parameters. I could never reproduce those impressive results in realistic out-of-sample tests of the published methods.

deteriorate soon after the training. But you can see that price curves produce often those predictable patterns that can be exploited in strategies. Those patterns do not exist in random data: The hit curve would have an arbitrary shape in such a case with no significant rising slope.

Another quite interesting inefficiency can be seen in the following two price histograms (Fig. 9):

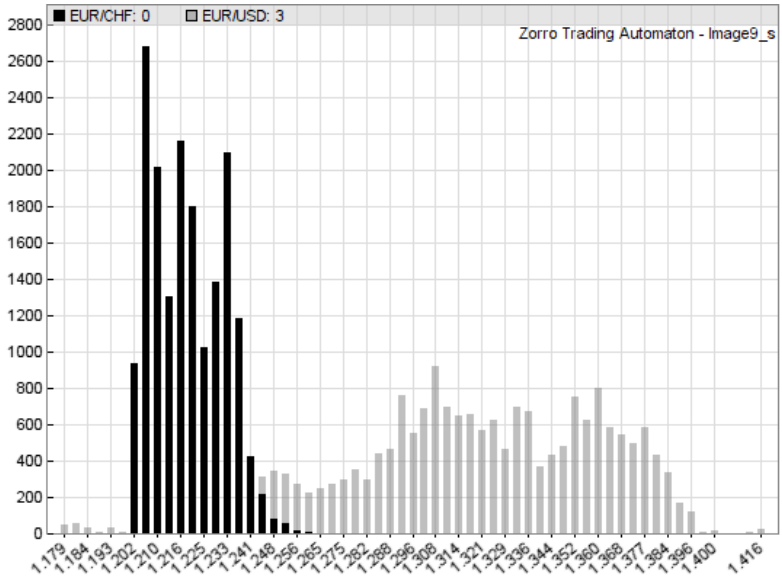


Fig. 9 - EUR/CHF and EUR/USD, price frequency distribution

The x axis is the price, the y axis the frequency of that price in the price curve. The dark bars are from the price of the Euro in Swiss Francs (EUR/CHF), the bright bars from the price of the Euro in US Dollars (EUR/USD). You can see that the EUR/CHF price distribution is rather narrow with a sharp border at the left. As opposed to the EUR/USD price distribution, which covers a much greater price range. This is how price distributions normally look. The compressed EUR/CHF distribution is an **anomaly** – and anomalies always indicate inefficiencies that can be used for profit in algorithmic trading. This one was caused by the 1.20 EUR/CHF price cap established by the Swiss National Bank from September 2011 until January 2015. The purpose was protecting the Swiss Tourism and export industry against an overvalued Franc. The price cap not only prevented the EUR from dropping below 1.20 Swiss Francs, it also worked against price movements in the opposite direction. Therefore the narrow distribution. This inefficiency does not

look very exciting, but has produced consistent and easy profits for algorithmic traders.

Of course, those are only a few of the many possible deviations of price curves from random curves. This sounds as if it were very easy to hack the financial markets. But trading is a game of probability. Most inefficiencies produce only a small edge, a win rate only slightly above 50%. Any tiny mistake immediately converts a winning system to a losing one. Any inefficiency that promises huge and safe profits is very visible in the price curve and often quickly identified and exploited by other market participants. This affects the market and eliminates the inefficiency sooner or later.

The mentioned EUR/CHF anomaly is an example of that. It was significantly visible and promised profits with little risk. The consequences can be seen in the following chart (Fig. 10):

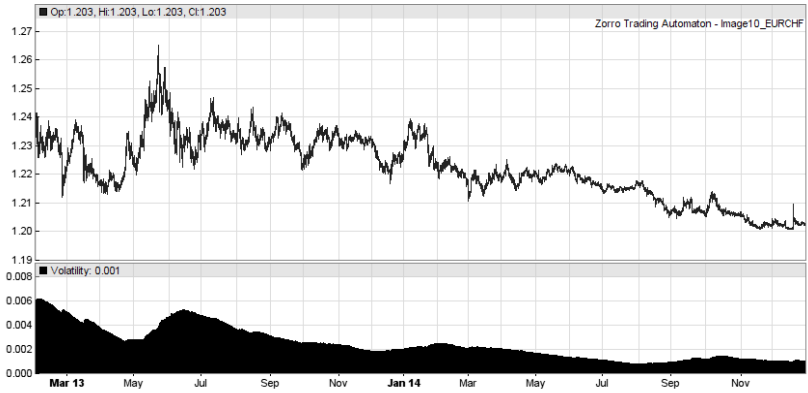


Fig. 10 - EUR/CHF exploitation by grid trading systems

The top window displayed the EUR/CHF price curve, the bottom window its **volatility**, i.e. the average amplitude of price movements. We remember that EUR/CHF had a significant inefficiency, the narrow price distribution caused by the price cap. At first a gang of hedge funds attempted to monetize it – however not with algorithmic trading. During 2012 they purchased CHF like mad. The hope was to push up the CHF price until breakdown of the price cap. This had caused an immediate price jump and some fine gains. But alas, it did not work this way. You do not mess with the Swiss National Bank. They used their inexhaustible money reserves for protecting the price cap with massive Franc selling. In 2012 they invested a total 200 billion Dollars

in protective selling. The hedge funds took a beating and withdrew by the end of 2012 with the tail between their legs (and probably painful losses).

Now the path was free for algorithmic traders. During the 2012 CHF battle they were forced to inactivity, since the price was nailed shut near 1.20. In January 2013 the first hackers started to exploit the market inefficiency with a specific method, a Grid Trader. This turned out a money-printing machine. The cap prevented the EUR/CHF from falling below 1.20, but it also prevented it from rising too high, since the SNB must eventually buy back all the Francs they have sold for defending the cap. Keeping prices inside a channel is an ideal case for a grid trading algorithm. Any price movement up or down produced profits almost without risk. More and more private traders and financial hackers, and also more and more large market participants jumped on the bandwagon. Three years after installation of the price cap, thousands of such systems sat on the EUR/CHF price curve like leeches and sucked off money. The result was a continuously falling price volatility. Which you can see in the lower part of Fig. 10. Lower volatility means lower profits to a grid trader. More capital must be invested, and the grid must be tightened for compensating. But there is a natural limit. You can not have a grid size smaller than the trading costs. By autumn 2014 the volatility was close to zero. You could not live from the EUR/CHF any more. And those who tried nevertheless had to invest massive money amounts for getting a profit out of the tiny price fluctuations.

It is well known what then happened to the Swiss Franc (Fig. 11):

Fig. 11 EUR/CHF crash

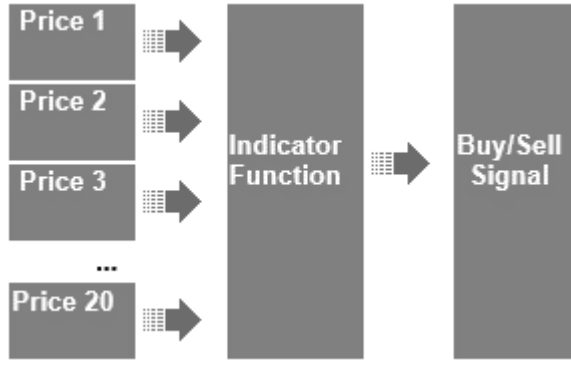
In the morning of 15 January 2015, the SNB gave a press conference and announced the cancellation of the price cap. The EUR/CHF fell in minutes like a stone from the 1.20 limit to below parity. Since many traders had invested large capital in the pair, the crash hit them all the harder. Even some brokers could not pull out in time and went belly up.

As we can see, trading systems have an expiration date. The more visible the inefficiency, the easier the profit, the sooner is it over. The life time of such specialized system is often only a few years. When developing trade strategies, you need not only be right, but also fast.

Technical analysis – sense and nonsense

The classical method of finding trade opportunities by analyzing past prices is named **Technical Analysis** (**TA** in short). It is based on the belief that a different way of representing the price curve – by so-called **technical indicators** - gives insight into the market situation and indicates **trade signals** for buying and selling at the right moments.

Normally you want to know if the price will rise or fall. For this prediction you look at the last part of the price curve - say at the close prices of the last 20 bars. These 20 price values must be reduced to a single number, for instance the future price trend. Mathematically, you want to get rid of 19 degrees of freedom, using a transformation from a 20-dimensional space into a 1-dimensional space. Such a transformation function is called a technical indicator.



Calculating with 20 dimensions sounds complicated, but most indicators are primitive. One of the most popular is the **Simple Moving Average** (**SMA**) It just adds all 20 prices and divides the sum by 20. The result is the average price of the last 20 bars. If the current price rises above the average, some traders believe that prices will further rise, and take this as a buy signal. It the price falls below the average, they believe that prices will further fall and they must sell. At least that's the theory.

Trade decisions based on indicators are the principle of technical analysis. In Fig. 12 you can see the SMA of the price curve from in Fig. 1 as a black line:

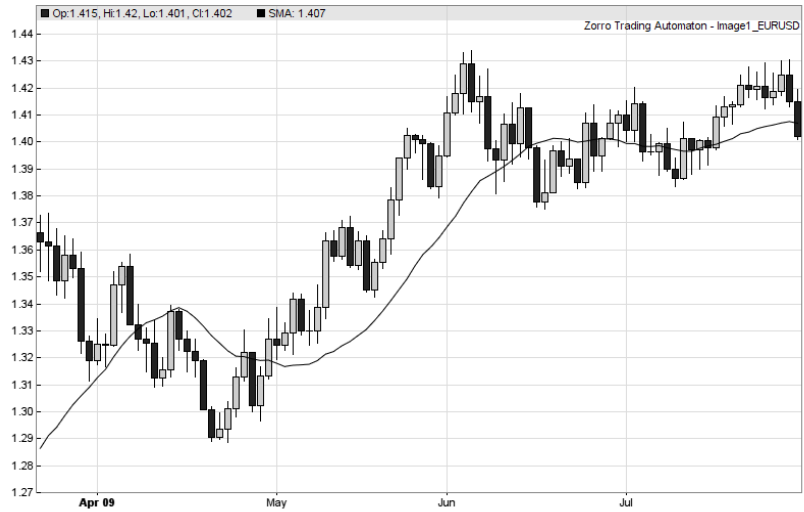


Fig. 12 - EUR/USD chart with SMA line

Of course, instead of the last 20 bars you can use any different number of bars or any time period for an indicator. You can differentiate between the high, low, open, and close prices of a candle, or use other data such as the market volume. And instead of averaging the prices, other indicators calculate their variance, their rate of change, their breakout from a given range, their maxima and minima of a given time period, and so on. Any such indicator contains some partial information from the price curve. It can be used to generate buy or sell signals when reaching a threshold or crossing another indicator. Because almost none of the classical indicators is based on a solid mathematical foundation or financial theory, any is as good as the other. And anyone can anytime invent new indicators, and anyone does. About 600 different indicators are meanwhile published in books and trader's magazines. For any arbitrary point in any price curve there are 10 indicators that recommend buying and 10 others that recommend selling. This might give you the

impression that something must be wrong with TA: If any one of the 600 indicators would really work and produce useful signals, there would be obviously no need for the other 599. Are technical indicators just garbage?

Traders often firmly believe in the predictive power of indicators, otherwise they wouldn't bet their money on them. Financial scientists normally don't believe in indicators and make fun of the uneducated traders. Hackers have no prejudice in any direction. They must examine the matter in detail first. What is predictivity, after all? Obviously it is not alone a property of an indicator. The random price curves of a perfectly efficient market cannot be predicted, no matter how intelligent the indicator. Price curves with some regularity, for instance sine curves, are predictable with almost all indicators. So the real question is: Are real price curves predictable with classical technical indicators, at least so far that profitable trading is possible?

This question is surprisingly hard to answer. The problem: When experimenting with indicators and their parameters, you can always find a combination that is profitable with some historical price curve. This simply results from the huge number of possible indicator and parameter combinations. If such results are evidence of a predictive power was first seriously tested in 2007 by Prof. D. Aronson of Baruch college[1]. His study involved thousands of trade rules with all classical indicator types and price, volume, and interest data series from 1980 to 2005. The results were adjusted by a bootstrap algorithm, named **White's Reality Check**, for eliminating data mining bias. In this study, none of the tested classical indicators came out with any predictive value. No matter with which data they were fed, they fared no better than flipping a coin.

However, this does not mean that technical indicators are completely worthless. Most indicators can temporarily work when a predictive price pattern develops within a limited market and time period. An example was the abovementioned Turtle Trading System that used such an indicator - the

[1] Described in D. Aronson, Evidence-Based Technical Analysis (2007)

Donchian Channel - for trade signals in the 1980's. This system was profitable for almost 10 years. Newer studies[1] found that indicators become predictive when their parameters – for instance, the number of bars whose prices are used for the calculation - are regularly adapted to the market situation. This can be done by a real time optimization, or by another adaption mechanism. In our example with the S&P500 cycles (Fig. 6), this would be an algorithm that adapts the bandpass filter permanently to the dominant cycle of the price curve. We will see in the next chapters how to use such methods.

Aside from the traditional indicators with their dubious value for generating trade signals, what else can we use? Simply science. Any particular inefficiency can be identified with precise mathematical solutions that differ from classical indicators like a scalpel from a rusty kitchen knife. If an asset shows regular swings upwards and downwards, a part of the price curve can be predicted with polynomial regression or Hilbert transformation. Spectral filters can remove noise and detect cycles or seasonality. Price curve patterns can be identified with the Fréchet algorithm that is otherwise used for handwriting detection. A perceptron can find price curve properties that are not visible to the human eye. If the inefficiency consists of a combination of independent features, it can be identified with a decision tree. All those methods are available with the software presented in the next chapter.

Another possibility is trading not technically, but fundamentally – deriving the prediction not from the price curve, but from other properties and dependencies of the traded asset. If the world is threatened by recession, the price of crude oil can be expected to drop, while the gold price will rise. Contrary to popular belief you can trade fundamentally with a computer algorithm. It only needs to analyze and follow the decisions of "informed market participants".

Conclusion

[1] P.H. Hsu and C.M. Kuan, Journal of Financial Econometrics 3, No. 4 (2007)

► Private financial trading is good for the global economy – but only when it produces profits.

► There's a large amount of deception and self-deception in the trading Nothing is as it seems.

► Algorithmic trading systems exploit market inefficiencies. There are various types of inefficiencies that can be identified with computer based analysis.

► Inefficiencies are temporary. Almost all trade strategies have unprofitable periods and/or an „expiration date".

► Technical analysis with classical technical indicators is not well suited for successful trading.

2 Programming in 3 hours

For hacking the financial markets, you need two things. First, some software that allows you to analyze, test and execute trade strategies. Second, a computer language to define your strategies. Of the hundreds of existing trade platforms and languages, only a few are suited for financial hacking. Visual "strategy-builders" or spreadsheet programs can only be used for very simple systems. For serious financial research or strategy development, programming skills are absolutely essential.

A programming language is much faster to learn than a foreign language, since you need not drill in some vocabulary. If you still want to avoid it at all costs, just skip this chapter and the sections with the code. But in a modern society, everyone should actually be able to understand code to a certain degree. Telling a computer what to do is required for any serious task that involves handling of data. The profession of programmer will probably die out sometime, just as the profession of writer in ancient civilizations: programming is a basic skill, like the ability to drive a car. On top of that it's fun. It may even become addictive. And even if you should never trade in your life, you can use this skill again and again - be it for the design of an interactive website, for time-saving Word macros or to mention it in your job application.

For defining trade strategies, you're not free to select the programming language with the simplest syntax, since it is often too slow for backtests and optimizations. Most algorithmic trading systems, as well as almost all professional software is based on the language **C** or its variants. The systems in this book are no exception. **C** and her sister **C++** are the fastest high-level computer languages and run a strategy simulation about 30 times faster than, for example, the popular Python. You will see later that language speed plays an essential role in trading strategy development.

Our hacker tool is **Zorro**, the Swiss Knife of trading. It is basically a frontend of a fast language with trading and data analysis functions. This allows it to trade, analyze data, plot diagrams (for instance all the diagrams in this book) and perform all sorts of financial tasks. For the examples in this book you'll

need Zorro version 2.06 or above that you can download here: https://zorro-project.com/download.php. Do that now.

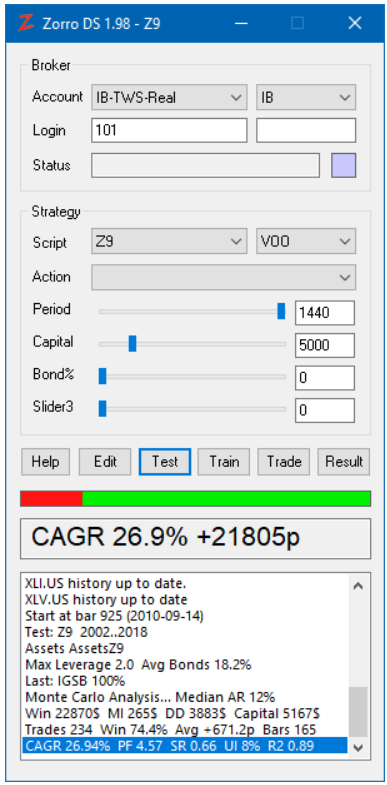


Fig. 13 Zorro

As you can see, the user interface is more functional than pretty. But it is designed for minimal hardware and space requirements, so it can run in trading mode on any small netbook screen in the background and you don't have to stare at it all the time. And if you still do, the displayed profit will hopefully make up for the austere design.

There is no menu. Almost everything is done by script. The user interface consists of four sections. The first, **Broker**, sets up the direct connection for automated trading. The fields are for selecting the account type and broker connection, and for entering your user ID and password. So if you open a demo account or a real account with a broker, you enter there the login data you got. Then you can start trading immediately. In the **Status** window you can see the server status of the broker connection. The small square next to it lights up green if a connection is established, and red when the connection

is interrupted, for example due to an Internet outage. Zorro then reconnects automatically when the broker is online again.

The second section **Strategy** determines Zorro's activities. With the **Script** scrollbox you can select one of the scripts from Zorro's **Strategy** folder. Scripts are text files containing computer instructions in the programming language C. They determine what Zorro shall do. The scroll box right of it allows to quickly assign an asset to the script, for testing its trading behavior or statistical properties. The **Action** scrollbox can be configured to to quickly start favorite programs, scripts, or script functions. The slider below determines the bar period in minutes. Three other sliders are for setting up script dependent parameters (in the image "Capital" and "Bond%", but they can have other names and meanings).

The buttons below cause Zorro to do something. **[Help]** opens the help file, **[Edit]** a text editor to edit the current script. **[Test]** runs the script. **[Train]** runs it several thousand times for finding optimal parameter values. **[Trade]** connects to the broker and start a trading session with the script. **[Result]** produces intermediate or final results as graphs and report files.

The lower section of the user interface includes a color bar, a text window and a larger window for messages. There you'll see everything Zorro has to tell you. The color bar and the text window can be controlled by the script.

After installation, play around with Zorro. Use the scroll box to select the script **Mandelbrot** and click **[Test]**. Then you should see something like this:

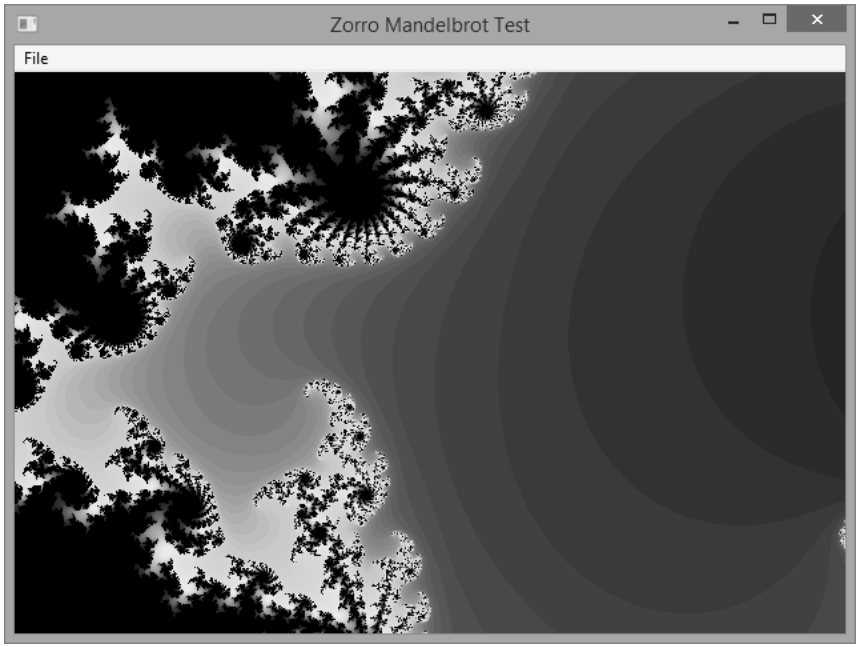


Fig. 14 Mandelbrot script

Actually this has nothing to do with trading, but it shows you that you can do anything with Zorro. It just depends on what is written in the script.

A **script** (also called **program** if it is a bit longer and more complex) consists of a list of instructions that tell the computer what to do under which circumstances and in which order. The instructions are formulated in a simplified language called **programming language**. In theory there could be other ways to define a trading system; for example, by a list of filters and indicators, or with some visual "strategy builder", or with formulas in a spreadsheet program. And some trading platforms indeed offer those ways for automated trading. But this limits the trading system to a set of simple, predefined rules - and this does not work well for achieving any profit in today's markets. All known successful strategies are based on scripts or programs.

The downside of a script is obvious: You have to learn a programming language. Zorro's lite-C is a version of C that 'hides' complex language elements and thus offers an easy way of writing scripts. In my experience, 90% of people learn the basics of programming quickly. But no one can tell beforehand whether you belong to the other 10%. Find out by trying.

On trader forums, scripts are often called "**Expert Advisors**", or "**EAs**". This has its origin in the marketing language of a widespread trading platform - the "MetaTrader" - with which you can also write scripts for trading systems. Its language, named "MQL4" or "MQL5", is also based on C, but unfortunately a lot more complicated to handle than lite-C. Since the "MetaTrader" has only rudimentary testing and analyzing functions, it is not really suited for financial hacking. However, you can control it with Zorro and this way use its broker connection for trading.

The content of a script, named **Code**, is normal text and can be edited with any text editor. It might look like this:

```
function run()
{
  vars Closes = series(priceClose());
  vars SMA100 = series(SMA(Closes,100));
  vars SMA30 = series(SMA(Closes,30));

  if(crossOver(SMA30,SMA100))
    enterLong();
  if(crossOver(SMA100,SMA30))
    enterShort();
}
```

This is a primitive (and not really profitable) trend following strategy. It uses two simple moving averages (SMA) with 30 and 100 bars time period, and buys a long or short position when the faster average crosses over or under the slower average. The meaning of the strange key words, parentheses, and semicolons will become clear soon.

If you already know C or lite-C, you can skip this part. If you already know interpreter languages like Python, C will appear a bit strange in comparison. But it has a crucial advantage for trading system development: It is much faster. You'll see soon that a script in C will need only a few seconds for a high-resolution backtest. The same script in Python would need hours.

Hour 1: Variables

A **script** (or **program**) is a list of **instructions** in plain text format that tell the computer what to do under which circumstances. It consists of two types of objects, **variables** and **functions**. A variable is a place in your computer's memory (just like a container) that is used to store numbers, text, or other

information. Because you don't want to have to remember where in the computer which variable is stored, a variable has a name that's used in the script. A few example script lines that define variables:

```
var Price;
var PercentPerMonth = 1.5; // the monthly interest
int Days = 7;
string Wealth = "I am rich!";
bool Winning = true;
```

These are a few short lines of code, but we learn many new things from them:

► Every variable must be **defined** (programmers also say '**declared**') before it can be used. We have different variable types[1]: **var** for a number with decimals, like prices; **vars** for a series of many numbers; **int** for a number that has no decimals, such as for counting something; **string** for text; and **bool** for a sort of 'toggle switch' that is either true or false. There are more basic variable types in lite-C, but in trading scripts you'll normally encounter only these five. If you write this line in your script:

```
Days = 3;
```

and you haven't defined the variable named **Days** before, you'll get an error message. The exception are variables such as **Spread**, **Stop** etc. that Zorro already knows because they are pre-defined in the language. There are about 100 such pre-defined variables that do not need to be declared as Zorro knows them already.

► Any variable can receive an initial value at start, but we aren't forced to do that when the initial value does not matter. Example:

```
int num_days = 356;
```

---

[1] For C/C++-programmers: **var** is **double**, **vars** is **double***, and **string** is **char***. They have been typedef'd for didactic purposes.

```
var limit = 3.5;
var X;  // initial value not relevant
```

► We can add **comments** to the code. Every time it encounters two slashes //, the language compiler will ignore the words that follow it, up to the end of the line. This way we can add useful information to our code:

```
int bar_width; // the width of one bar in minutes
```

or we can temporarily disable a line in the script by placing two slashes in front of it. This is called "**commenting out**" a line, and is normally used so frequently that the script editor has two extra buttons for commenting out and commenting in.

► Every definition or any command in C needs to end with a semicolon. Many beginners forget to add the nondescript ";" at the end of their lines of code, and this also leads to an error message - not in the line with the missing semicolon, but in the following line!

► Every variable name must start with either a letter or with an underscore _, and most not contain any other special characters. Here are some valid variable names:

```
var AlohA;
var _me_too;
var go42;
var Iam19;
var _12345;
```

Some bad examples:

```
var “I told You”;
var 1_for_all;
var 12345;
```

I'll let you discover what is wrong in the definitions above.

► Variable names are case sensitive. This means that if we define an **int** this way:

```
int MyTradePositions;
```

and then we use it later in our code this way:

```
mytradepositions = 5; // or
Mytradepositions = 5; // or
MYTRADEPOSITIONS = 5;
```

Zorro will not accept it.

► You can define several variables of the same type in one line. This saves lines and keeps your code short. Example:

```
var momentum, strength, score;
int width = 7, height = 14, depth = 20;
```

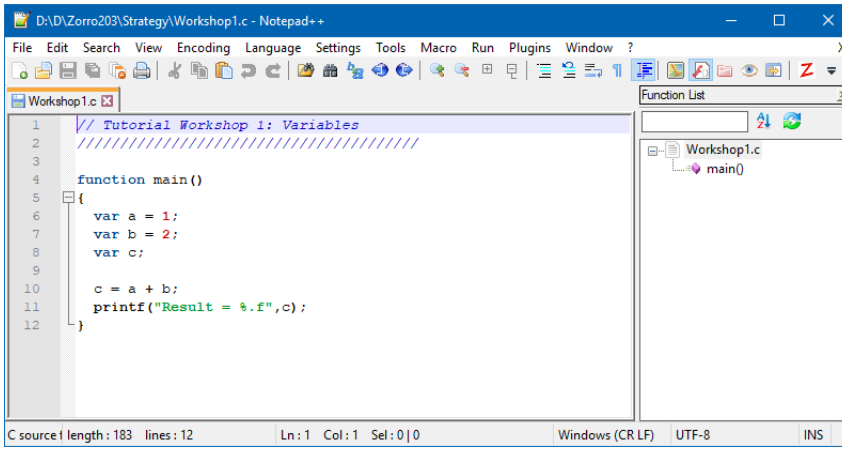
► Finally, the variables should have significant names. While it is possible to define a pile of variables like this:

```
var x32;
var number125;
var h_34_5;
var _z34187;
```

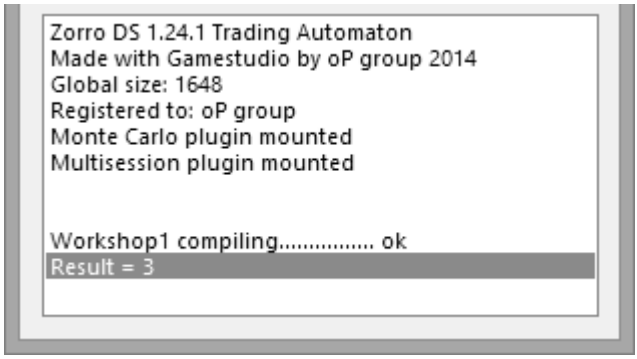
it isn't a good idea to do it this way. You will have problems trying to remember what these variables do if you look at your script a few weeks later. If you want to show your script to other people, they will also have a hard time trying to figure what you wanted to do with your code.

The pocket calculator

Enough theory. Let's play around with these variables. Start Zorro, select **Workshop1** in the Script dropdown menu, then press **[Edit]**. The script editor will open up and show you this script:



Now, press [**Test**] on Zorro's panel, and watch what happens in its message window:

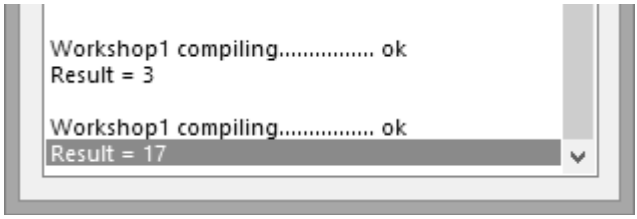


By the way, what does „compiling“ mean? The **Compiler** is a part of Zorro, used for translating the script into machine code that can be directly executed by the Pentium processor inside your PC. This happens next, and the result is printed to the following line: **Result = 3**.

Now, edit the script in the editor. Replace the a and b variable definitions with

```
var a = 5;
var b = 12;
```

Save the edited script (**File/Save** or **[Ctrl-S]**), then press **[Test]** again:



Seems to make sense? It looks like **c** is the sum of **a** and **b**. Now let's take a look at this miraculous script that transforms Zorro in a sort of adding machine.

```
// Tutorial Workshop 1: Variables
//////////////////////////////////////

function main()
{
  var a = 5;
  var b = 12;
  var c;

  c = a + b;
  printf("Result = %.f",c);
}
```

The script begins with a comment (the lines with **//**) that tells us what it is. Then we have a **function** named **main** - everything that happens in a C program is inside the winged brackets **{ }** of functions. But we'll come to functions in the next workshop. Here we concentrate on variables:

```
var a = 5;
var b = 12;
var c;
```

Three simple **var** definitions as described above. The following line is the core of our script:

```
c = a + b;
```

This line of C code appears to make **c** equal to the sum of **a** and **b**. In fact it is a **command** (also called **instruction**) to the computer to add the content of the variables **a** and **b**, and store the result in the variable **c**. Commands are lines in the code that usually do something, for instance alter a variable.

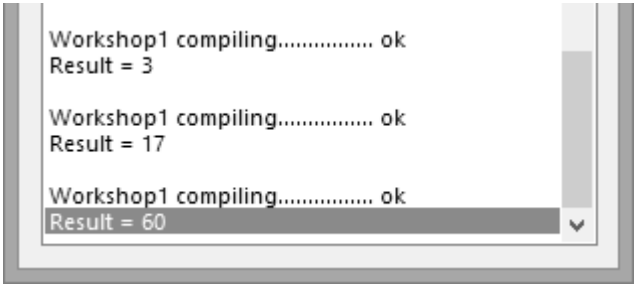
The last line is also a command, used for displaying the content of **c** in the message window:

```
printf("Result = %.f",c);
```

Let's make an experiment. Find the line of code **c = a + b;** in the editor, and then replace the "+" with a "*", the 'times' character, so that the line now reads:

```
c = a * b;
```

Save the script, then press **[Test]** again:



We have now done the first steps with the C programming language. Zorro has multiplied **5** by **12**, displaying the correct result. It's not yet a trade strategy what we're doing here, but we're getting closer. So now we know how to add and multiply values; we can use "-" to subtract two numbers or "/" to divide them. However, the second important element of a programming language is yet missing: functions.

Hour 2: Functions

Understanding variables and functions means understanding how a script works. Just like variables, functions must be declared before they can be used. Here's an example of a function declaration:

```
function add_numbers()
{
  var a, b, c;
  a = 3;
  b = 5;
  c = a + b;
```

```
}
```

A function is nothing more than a collection of C commands that are executed by the computer one after the other. Let's see some properties of these functions:

► A function is normally defined using the word **function**[1] followed by the name of the function and a pair of parentheses **( )**. The parentheses are used to pass additional variables to the function; more about that later. In this case we don't pass any variables, so they are empty.

► The **body** of the function - its list of commands - must be written inside a pair of winged brackets **{ }**. The body consists of one or more lines of C code that can contain variable declarations or commands, and end with a semicolon. For clarity, programmers usually indent the code in the function body by some spaces or a tab, for making clear that it is inside something. Spaces and tabs are ignored by the compiler, so they can be used to beautify the code.

► The names used for functions follow the same naming convention as for variables. And you shouldn't use the same name for a variable and a function. In Zorro scripts we often follow the convention to begin a variable with an uppercase letter and a function with lowercase; but this is totally up to you.

If you want to hack the financial markets, I hope you can quickly and without pen and paper convert 500 Euros to Dollars at a rate of 1.36. What, you can't? Then let's write a function for this. We write first the word **function** and then the name of the function; let's name it **euro_to_dollar**:

```
function euro_to_dollar()
{
```

[1] For C/C++ programmers: **function** is an **int** function, but returning a value is optional.

Quick check: We haven't forgotten the parentheses after the name of the function, and the first curly bracket is also in place. We will use some variables inside the function, so we better define them now:

```
  var Euro = 500;  // Amount of Euro for converting
  var Rate = 1.36;
```

Nothing new so far. We have defined two **var** variables and they have received initial values. Now comes the scary part: how will I be able to tell the computer to calculate the dollars? How would I do it with a pocket calculator? I would enter something like this:

500 * 1.36 =

When we use variables instead of numbers, it looks like this:

Euro * Rate

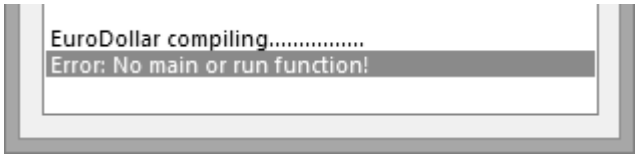
Ok, so our function should end like this:

```
  var Dollar = Euro * Rate;
  printf("%.f Euros are %.f Dollars.", Euro, Dollar);
}
```

We've remembered to add the second bracket, so now the body of the function is enclosed by the two required curly brackets. Now let's test it. Fire up Zorro, and then select **[New Script]** in the Script list. Wait until the editor opens with an empty script. Then enter the lines below:

```
function euro_to_dollar()
{
  var Euro = 500;  // Amount of Euro for converting
  var Rate = 1.36;
  var Dollar = Euro * Rate;
  printf("%.f Euros are %.f Dollars.", Euro, Dollar);
}
```

Now save the script (**File/Save As**) in Zorro's **Strategy** folder under a name ending with „**.c**", such as „**EuroDollar.c**". If you did everything right, you should now find **EuroDollar** in the script scrollbox. Select it. Time to **[Test]** our script. But what is that?

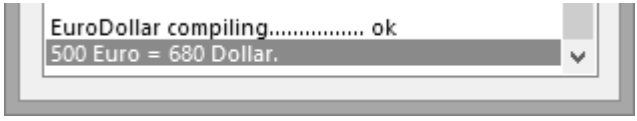


Does this error message mean that a script always needs a **main** or **run** function? It is indeed so; **main** is a special function name. If a function is named **main**, it will automatically execute when we start our script. The function named **run** is special to Zorro; it contains our trade strategy and is automatically run once for every bar period. If a script has neither a **main** nor a **run** function, Zorro assumes that you made a mistake and will give you this error message.

Now, let's give the script what it needs and add a **main** function at the end of the code:

```
function main()
{
  euro_to_dollar();
}
```

This **main** function is obviously quite short – it contains only a single command for executing – or "calling" as the programmer says – the function **euro_to_dollar**. For calling a function we write its name followed by a pair of parenthesis and then the usual semicolon. The proof:

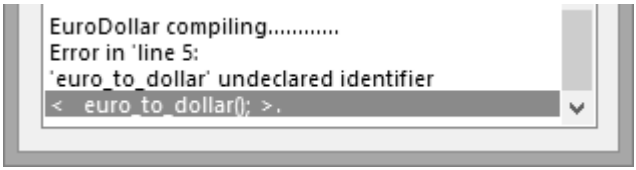


Using functions, you can add new commands to a programming language, just like inventing new words for a human language. Important tip: write the code for your functions above the code that calls them. The computer reads code the same way you read a book: it starts with the top of the script page and goes down to the bottom, reading the code line by line. If we would write the script the other way around, like this:

```
function main()
{
  euro_to_dollar();
}
```

```
function euro_to_dollar()
{
  ...
}
```

the compiler will say: oh, that's function **main**. I need to run it first. What does it say now? Call **euro_to_dollar()**? What's that? I can't call what I don't know. I'm going to display an error message and will take the rest of the day off.



You will encounter compiler errors frequently when you write scripts - even experienced programmers make such mistakes all the time. Sometimes it's a forgotten definition, sometimes a missing semicolon, an extra parenthesis or a missing bracket. Get used to compiler errors and don't be helpless when you see one. The computer (usually) tells you what's wrong and at which line in the script, so you won't need rocket science for fixing it.

A short summary of what we've learned:

► We define simple functions by writing "**function name() { ... }**".

► If a function has the name **main** or **run**, it is automatically executed. All other functions must be called from an already-running function to be executed.

► Don't panic when you get compiler errors - they happen all the time and are normally easy to fix. It is uncool to ask on a forum for help with compiler errors!

Passing variables to and from functions

A function can also get variables or values from the calling function, use them for its calculation, and give the resulting value back in return. Let's see an example of a function that gets and returns variables:

```
var euro_to_dollar(var Euro)
{
  var Rate = 1.36;
  return Euro * Rate;
}
```

The **var Euro** between the parentheses is for passing a value to the function. This value is stored in the variable **Euro**, which can be used inside the function like any other variable. For returning a value from a function, just write it behind a **return** command. It can be a number, or a variable, or any arithmetic expression that calculates the returned value. The function then gives that return value back to from where it was called.

You did notice that we defined this function not with the keyword "**function**", but with "**var**"? But is **var** not a variable definition? Yes, but when a function is expected to return something, it must be defined using the type of the returned variable. So if a function returns a variable of type **int**, define it with **int** instead of **function**; if it returns a **var**, define it with **var**. The compiler will still know from the **(..)** parentheses that this is a function definition and no variable definition.

If a function expects variables, put their definition - type and name - in the function definition between the parentheses. The space between the parentheses is named the **parameter list** of the function. If there are several variables in the parameter list, separate them with commas. When you then call that function, just put the values of the variables you want to pass to the function between the parentheses. The values passed to the functions are named the **arguments** of the function.

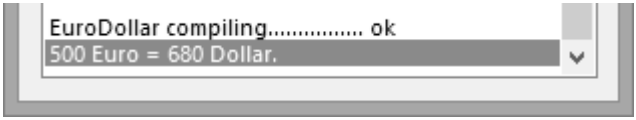
If a function returns something, you can just place a call to that function in stead of the value that it returns. This sounds sort of complicated? Let's try it right away with our new **euro_to_dollar** function. This is our new script:

```
var euro_to_dollar(var Euro)
{
  var Rate = 1.36;
  return Euro * Rate;
}

function main()
{
  var Euro = 500;  // Euro amount for conversion
  var Dollar = euro_to_dollar(Euro);
  printf("%.f Euros = %.f Dollars.", Euro, Dollar);
```

```
}
```

This makes our code shorter and more elegant. Still, the result is the same:



Note that **Euro** is a variable inside the **main** function, but a parameter inside the **euro_to_dollar** function. Both variables don't know of each other: a variable defined in a funciton only exists within that function. But both have the same value, as the **Euro** parameter gets its value from the **Euro** variable when the function is called.

The printf function

Now finally, what is this mysterious **printf(..)**? It has parentheses attached, so it's obviously a function that we call for displaying our result. However we have nowhere defined this function; it is a function that is already "built-in" in C. Just as the built-in variables that we mentioned in the last workshop, there are also many functions already built in the script language.

Other than normal functions, **printf** accepts any number of parameters, even of different types. We pass three parameters here, separates by commas:

```
"%.f Euros = %.f Dollars.", // first parameter
Euro,  // second parameter
Dollar // third parameter
```

The first parameter is a **string**, used for displaying some text: "**%.f Euros = %.f Dollars.**". The second and third parameters are of type **var**: **Euro** and **Dollar**. You can read details about the **printf** function in the Zorro manual or in any C book; for the moment we just need to know that the strange "**%.f**" in the string is a placeholder. It means: the function inserts here the value - with no decimals - of the next **var** that is passed to the function. So, if **Euro** has the value 500 and **Dollar** has the value 680, our **printf** function will print "**500 Euros = 680 Dollars.**".

We can make our code even shorter. Remember, if a function returns a **var**, we can just place a call to this function in stead of the **var** itself - even inside the parentheses of another function. We'll save one variable and one line of

script this way. Programmers do such shortcuts all the time because they are lazy and prefer to type as less code as possible:

```
var euro_to_dollar(var Euro)
{
  return Euro * 1.36;
}

function main()
{
  var Euro = 500;
  printf("%.f Euros = %.f Dollars.", Euro, euro_to_dollar(Euro));
}
```

Here's an important tip when you call functions in your code that return something. Do not forget the parentheses - especially when the parameter list is empty! In C, a function name without parentheses means the address of that function in the computer memory. **euro_to_dollar** and **euro_to_dollar()** are both valid code and won't give a compiler error message! But they are something entirely different. For avoiding errors, Zorro's 'built-in' functions normally begin with a lowercase character (such as **enterLong()**) and all built-in variables begin with an uppercase character (such as **BarPeriod**).

In the next hour we will see how a script can make decisions (called 'branches' in computerish). Being able to decide when to buy and when to sell is important for trading. So we're now going in huge steps towards our first trade strategy.

Hour 3: Branches and loops

If my monthly bills grow bigger than $3000, I need to program a new trading strategy, else I can continue my old strategy.

That was just an example of if/else branching; its associated code would look like this:

```
if(Bills > 3000)
  make_new_strategy();
else
  continue_old_strategy();
```

You will use **if** statements whenever you want your script to make decisions - meaning that it behaves differently depending on some conditions, like user input, a random number, the result of a mathematical operation, a crossing of two indicators, etc. Here's the basic form of the **if** statement:

```
if(some condition is true)
    do_something(); // execute this command (a single command!)
```

or

```
if(some condition is true)
{
    do_something();
   do_another_thing(); // execute one or several commands
}
```

A more complex form of the **if** statement is listed below:

```
if(some condition is true)
{
    ... // execute one or several commands
} else {
    ... // execute one or several commands
}
```

The instructions placed inside the **else** part are executed only if "**some condition**" is not true. As in our first example:

```
if(Bills > 3000)
  make_new_strategy();
else
  continue_old_strategy();
```

Either **make_new_strategy** or **continue_old_strategy** is called here, but never both since only one of them will be executed. The conditional parts of the code are called "branches" because the code can take one or the other way, and because because several nested if instructions can look like a tree with the root at the first "**if**" and many branches of which only one is executed.

By the way, have you noticed how we indented the lines after the "**if**" and between the winged brackets? The C language does not care how you indent lines or if you write a command on the same or on a new line, but we do that

for clarity. The code reads easier if dependent lines are indented. However if we wanted, we could write it also like this:

```
if(Bills > 3000) make_new_strategy();
else continue_old_strategy();
```

or even like this:

```
if(Bills>3000) make_new_strategy(); else continue_old_strategy();
```

Let's draw some conclusions:

► "if" branching statements start with the **if** keyword followed by a pair of parentheses;

► the parentheses can contain a comparison, or any other mathematical expression that can be true or false;

► if the expression is true, the following instruction or the set of instructions placed inside the first pair of curly brackets is executed;

► if the expression is false and we don't use "**else**", the set of instructions placed between the curly brackets is skipped (it isn't executed);

► if the expression is false and we are using the "**else**" branch as well, the set of instructions placed inside the first pair of curly brackets is skipped, and the set of instructions placed inside the second pair of curly brackets is executed.

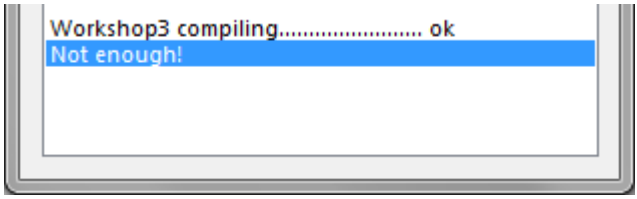
► indentation and lines are not needed for the language, but make code easier readable.

Enough theory – let's see some action. As in the following script:

```
function main()
{
  var Profit = 50;
  if(Profit > 100)
    printf("Enough!");
  else
    printf("Not enough!");
}
```

We have defined a **var** named **Profit** which receives an initial value of 50, and an **if** statement. If **Profit** is greater than 100, we have enough, otherwise not. We can omit the if/else pairs of curly brackets mentioned above when their content consists of a single line of code.

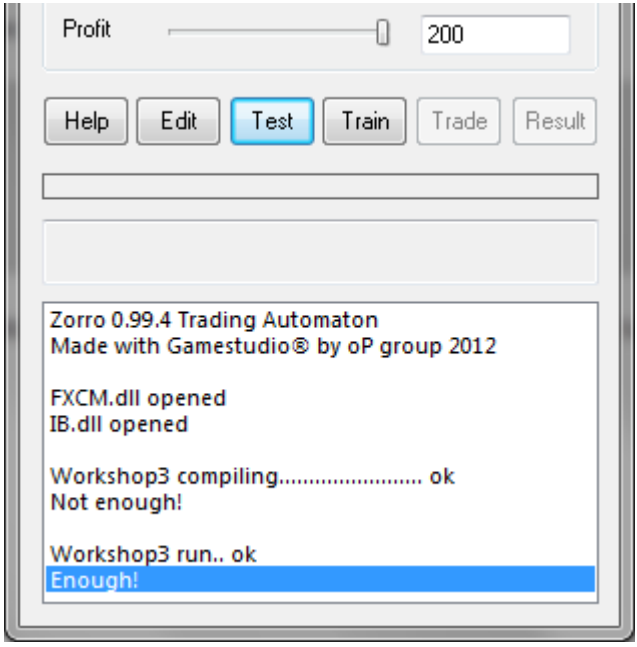
Create a new script - you have learned in the last workshops how to do that – write the content, save it in the **Strategy** folder, select it, and **[Test]** it:



Now let's try something else. Modify the code by editing the "**Profit**" definition line so that it now looks like this:

```
var Profit = slider(3,50,0,200,"Profit",0);
```

When you now click **[Test]** to run the script again, you'll notice that the bottom slider gets the label "**Profit**". Move it all the way to the right, so that **200** appears in the small window, and click **[Test]** again:



What happened? The **slider()** function put its return value - which is the value from the bottom slider - into the **Profit** variable, and thus the **if(..)** condition became true as the value was now bigger than 100. We can make an educated guess how the slider function works: It gets six variables - the slider number (**3**), the initial value (**50**), the right and left borders (**0** and **200**) of the value, the name of the slider ("**Profit**"), and a tooltip (here **0**, i.e. not used). Put the slider to the left again and verify that the program now prints "Not enough!" when you click **[Test]**, since the **slider** function now returned a value less than 100. You can now imagine how we can use the sliders for adjusting variables for our strategies.

Now let's assume that you want to do something only when two different conditions are fulfilled. Try the following program:

```
function main()
{
  var Risk = slider(2,50,0,200,"Risk",0);
  var Profit = slider(3,50,0,200,"Profit",0);
  if((Profit > 100) and (Risk == 0))
    printf("Enough!");
  else
    printf("Not enough!");
}
```

Now two sliders are involved. How do you need to set them for the "Enough!" condition? You can certainly find that out... and we've learned that we can combine two conditions using the "**and**" keyword. That means both conditions must be true for the combination to be true. There is also an "**or**" keyword when only one of the conditions needs be true.[1]

[1] For C/C++ programmers: the usual operators **&&** (and) and **||** (or) can of course also be used in lite-C.

Now, three important tips for avoiding coding mistakes in expressions and comparisons.

► Have you noted the use of parentheses around **(Profit > 50)** and **(Risk == 0)**? We know from school mathematics that in a mathematical equation, the expressions in the parentheses are solved first. (1+2)*3 is not the same as 1 + (2*3) - this is true in mathematics and also in a programming language. Always use parentheses to make sure that the program calculates in the same order that we want... and make sure that you have as many opening as closing parentheses! A missing parenthesis at the end of a line is one of the most frequent reasons of compiler error messages. The computer will usually complain about an error in the following line because it's there looking for the missing parenthesis. Or even about an error at the end of the script. That was the first tip.

► What's with that "**Risk == 0**" in the first new line of code? Is the double equals character a typing error? No, it isn't. Whenever you compare two expressions (**Risk** and **0** in the example above) you have to use "**==**" instead of "**=**", because a line of code that looks like this:

```
if(Risk = 0)
{
   ... // do some stuff
}
```

will set **Risk** to zero instead of comparing **Risk** with zero! This is one of the most frequent mistakes; even an experienced programmer might set a variable to a certain value by mistake, instead of comparing it with that value. Using one instead of two equality signs for comparing two expressions is a very frequent mistake. Don't forget this!

```
if(a == 3) // correct
{
   ... // do_some_stuff
}

if(a = 3) // wrong!
{
   ... // do_some_stuff
}
```

You can avoid this mistake is you make it a habit to put the constant value on the left side of a comparison and the variable on the right side. The **Risk** comparison statement would look this way:

```
if(0 == Risk) // correct
    ...
```

If you then accidentally put a single '=' instead of '==', the compiler will report an error because it knows that **0** can't be set to a different value. That was the second tip.

► The third tip is that you should use the '==' comparison with care when comparing **var** variables. If a computer calculates a mathematical expression, the result is usually inaccurate due to the limited floating point precision. Instead of 0.333333333..., dividing 1 by 3 results in 0.3333329856. Thus, a comparison will always come out false. When comparing values with many decimals, such as prices, it's highly unlikely that two such values will be ever exactly equal. Normally you'll only use 'greater' or 'smaller' comparisons (< or >) with **var** variables, except in special cases, for instance when you compare them with 0. This problem does not affect **int** variables, as they have no decimals and can indeed be exactly equal.

Loops

While I'm no millionaire yet, I must continue trading

That was just an example of a "while loop". In code it would look like this:

```
while(MyMoney < 1000000)
   trade(MyMoney);
```

The **while** statement has a similar syntax as the **if** statement. There is some expression - in this case the condition that the variable **MyMoney** is less than a million - and a command that is executed when the expression is true. However, in the case of an **if** statement, the program whould then go on with executing the next command afterwards. In the case of a while statement, the program "jumps back" to the **while** condition and tests it again. If it's still true, the command is executed again. And the while condition is then tested again. And so on. This process is repeated until the **while** expression eventually comes out false, or until eternity, whichever happens first.

A "loop" is called so because the program runs in a circle. Here's the basic form of the while loop:

```
while(some condition is true)
    do_something(); // execute this repeatedly until the condition
becomes false
```

or

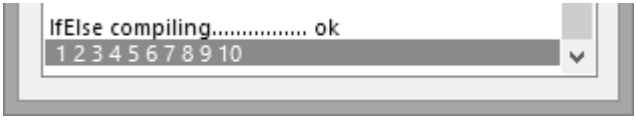
```
while(some condition is true)
{
    ... // execute all commands inside the curly brackets
repeatedly until the condition becomes false
}
```

Note that whatever the commands do, they must somehow affect the **while** condition, because when the condition never changes, we have an "infinite loop" that never ends!

Here's a practical example of a while loop. Run this in Zorro:
:

```
function main()
{
  int n = 0;
  while(n < 10) {
    n = n + 1;
    printf("%i ", n);
  }
}
```

This little program adds **1** to the variable **n**, prints the variable to the message window, and repeats this until **n** is 10.



And here a small puzzle for seeing if you paid attention. The **while** condition is true as long as **n** is smaller than 10. Bei n at 10 it is therefore false and the

following command is not executed. Why is still the 10 printed as the last number?[1]

C has different kinds of loops. The second-most frequently used is the **for** loop, normally used for counting something:

```
for(initialization; comparison; continuation)
{
  do_something ();
}
```

There are now three commands inside the parentheses. **Ininitalization** is executed once at the start of the loop. **Comparison** is checked before every loop execution and aborts the loop if false. And **continuation** is executed at the end of every loop. The **for** loop allows to formulate the above example a bit more elegant:

```
function main()
{
  int n;
  for(n = 0; n < 10; n = n + 1)
    printf("%i ", n);
}
```

Loops are very useful when something has to be done repeatedly. For instance, when we want to execute the same trade algorithm several times, each time with a different asset. This way we can create a strategy that trades with a portfolio of assets.

By the way, haven't I told you that programmers are lazy? Since adding 1 is very frequent in scripts, you can do it in C in three ways:

```
N = N + 1;
N += 1;
```

---

[1] The script adds 1 and prints the number afterwards. So the printed number is 1 more than the number in the **while** condition.

```
N++;
```

The three lines do the same – they increase **N** by **1** – but for obvious reasons, programmers like the third method the most.

We are now at the end of the basic lessons about general programming. In the next chapters we'll develop trade strategies; you've already learned all programming stuff that you'll need. But there's far more to programming - we haven't touched yet concepts such as macros, pointers, arrays, structs, classes, or the Windows API. lite-C also supports some elements from C++, the 'big brother' of C - such as methods or function overloading.

If you want, you can learn 'real' programming beyond the scope of trade strategies. Buy a C book or go through a free online C tutorial, such as "Sam's Teach Yourself C in 24 Hours". You can also join the Gamestudio community that uses Zorro's lite-C language for programming small or large computer games. Programming can be a lot of fun, even without earning money with it.

Conclusion

► Zorro is a hacker tool for analyzing price curves or other data sets, and for developing, testing, and running trade strategies.

► A script is for describing a task to a computer in a formalized language. It contains a list of commands.

► Script elements are variables and functions. Functions contain the commands, variables the data.

► Scripts can go far beyond developing trade strategies.

3 Follow the trend

Bob has traded Forex and CFDs since 1995. In some years his profits have been in the six figures area, and his only concern was the tax declaration. But lately it has not gone so well. In fact, he has been making losses for three years now. The reserves melt. Something must change. But what? Bob has already paid for lots of trading seminars. He has read lots of trading books. They didn't help. Only one option remains. And for this reason Bob has an appointment with Alice today.

Alice solves all kinds of problems that can be solved with a computer. Bob does not longer trust his trading instinct. He wants to replace himself with a computer program. For this, however, he has to explain to Alice how such a program should behave. She knows a lot of computers, algorithms, and data analysis, but she has no clue of trading.

Bob: "I'm going with the trend. I buy long when the price starts to rise. I'll buy short when it starts going down. "
Alice: "And this works?"
Bob: "Sometimes. Depends on the market. "
Alice: "So you buy long, for example, when today's price is higher than yesterday's price?"
Bob: "Nah, one single higher price alone won't do. Prices wiggle a lot. I just look at the long-term trend, let's say the trend of the last month. For this I take a moving average. That is, I take all the prices of the last four weeks and calculate their average. My trading platform does this by itself and shows me the average as a red line. The moment this line turns upwards, I buy. "
Alice: "Good. Should not be a problem to program that. "
Bob: "Well, actually there is a problem. You see, a moving average is not very good for trend finding. For getting a smooth trend curve the moving average period must be made long. But making the average long also lets it lag a lot behind the prices. The signals are just not timely for a good trade. The trend is already over when my moving average finally bends up or down. You need to sort of look ahead of the moving average curve, if you get my meaning."
Alice: "So you want to know when a 4 weeks trend changes, but you need to know it in less time than 4 weeks?"
Bob: " You got it."

Alice: "I could use some sort of lowpass filter instead of a moving average to smooth the price curve. For instance, a Laguerre filter. It has almost no lag. Will that be ok for you?"
Bob: "I do not know about those lowpass filters. But I trust you."
Alice: "You know them from your stereo. Lowpass filters remove high frequencies and keep only the low ones. For a price curve, I think this would be the trend."
Bob: "If you say so."
Alice: "And the program should always buy when the trend changes direction? Long, when it is at the low point and starts to rise, and short when it starts falling from a high point? "
Bob: " You got it."
Alice: "And when shall the positions be sold again?"
Bob: "At the right time. Depends on the market."
Alice: "I could close a previous long trade when opening a short trade, and vice versa. Does this make sense?"
Bob: "Yeah, I usually do that. If I have not been stopped out before. "
Alice: "Stopped out?"
Bob: "Sure. We need a stop loss. If the loss of a trade becomes too great, we must sell. You do not want my whole account wiped out because of a few bad trades? "
Alice: "Not before I got paid. At which loss shall the program close the trade?"
Bob: "This is a bit delicate. The prices go up and down all the time. If we stop the trades too early, they have no chance to follow the trend. If we stop too late, we get bad losses. "
Alice: "Let me guess: It depends on the market?"
Bob: "You got it."

A basic lowpass filter strategy

In this and the following chapters, we'll look over Alice's shoulder while she develops many different trading strategies. The strategies themselves are not this important. But the development process is. On the way we'll learn important details of scripting that were not covered in the programming course, and get introduced in the concepts, tips, tricks, and traps of system development. So follow Alice closely and reproduce all her steps. All the strategies and scripts can be downloaded from **financial-hacker.com**.

This is her first trading strategy script for Bob, based on a lowpass filter (**Alice1a.c**, at $1000 agreed development fee):

```
function run()
{
  vars Prices = series(price());
  vars Trends = series(Laguerre(Prices,0.05));

  Stop = 10*ATR(100);
  MaxLong = MaxShort = -1;

  if(valley(Trends))
    enterLong();
  else if(peak(Trends))
    enterShort();
}
```

That's already the complete system. We can see that the function is now named '**run',** not '**main**'. '**run**' is also a special function name, but while a **main** function runs only once, a **run** function is called after every bar with the period and asset selected with the scrollbars. By default, the bar period is 60 minutes. So this function runs once every hour when Zorro trades.

We're now going to analyze the seven code lines step by step. At the begin we notice two strange lines that look like **var** definitions:

```
vars Prices = series(price());
vars Trends = series(Laguerre(Prices,0.05));
```

However these special variables are defined with '**vars**' and initialized with a **series()** function call. We define not a single **var** here, but a **time series** - a **var** with a history[1]. A time series contains many single **var** elements. The first one contains the current value of the variable, the next the value the variable had one bar period before, followed by the value from two bar periods before and so on. Series are mostly used for analyzing price curves and other data curves. For instance, we could use a series to take the current price of an asset, compare it with the price from the previous bar, and do some other

[1] For C/C++ programmers: the type **vars** is a **double*** pointer.

things dependent on past prices. Indicators and similar functions take series as parameters.

For getting the current value of a series, add **[0]** to the series name; for the value from one bar before add **[1]**, for two bars before add **[2]** and so on. In Alice's code above, **Prices[0]** would be the current value of the **Prices** series, and **Prices[1]** the value from 60 minutes ago. All better trading script languages support time series in some way, and all indicator, statistics, and other financial functions use series for calculating their results. We'll encounter price series very often in trade scripts, so we have to get familiar with them.

The **series** function converts a single variable to a series. The variable or value for filling the series is normally passed as parameter to that function. However, we're not using a variable here, but the return value of another function call instead:

```
vars Prices = series(price());
```

This line defines a **var** series with the name "Prices" and fills it with the return value of the **price** function. As we've learned in the last workshop, you can this way 'nest' many function calls, using the return values of functions as parameters to other functions. By the way, Alice has the habit to add a 's' at the end of a series name for distinguishing series from normal variables.

The **price** function returns the mean price of the selected asset at the current bar. There are also **priceOpen**, **priceClose**, p**riceHigh** and **priceLow** functions that return the open, close, maximum and minimum price of the bar; however, the mean price is often the best for trade strategies. It's averaged over all prices inside the bar and thus less susceptible to random noise.

By the way, you can easily see what the built-in functions are used for. If you move the cursor to a function name in the script editor and hit **[F1]**, the electronic manual will pop up with a description of the function. It's the same manual that appears when you click on Zorro's **[Help]** button. It lists all functions and predefined variables and allows searching for keywords.

Next code line:

```
vars Trends = series(Laguerre(Prices,0.05));
```

Here we got again a nested call. The line defines a series named '**Trends**' and fills it with the return value from the **Laguerre** function, which is in turn called with the the previously defined **Price** series and the number 0.05. As

you probably guessed, this function is Alice's Laguerre lowpass filter. Its parameters are a data series and a smoothing parameter, which Alice has set to 0.05. The filter attenuates all the short term wiggles and jaggies of the **Prices** series, but it does not affect its trend or long term cycles. Its return value is the filtered price.

With such **spectral filters** - functions which remove or amplify a particular range of the frequency spectrum of a price curve - several types of inefficiencies can be identified in price curves. Filters of the second order have a steeper filter curve, which in practice means better attenuation and lower lag. For instance, the well-known indicator 'EMA' (**E**xponential **M**oving **A**verage) represents a low-pass filter of the first order. Fig. 15 shows the behavior of some of Zorro's spectral filters:

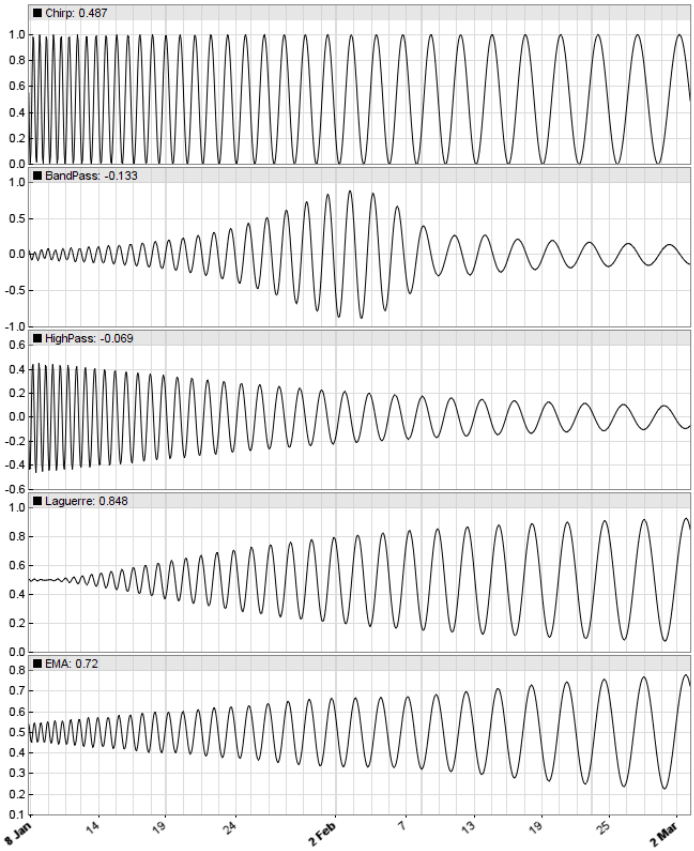


Fig. 15 – Some spectral filters

The first curve at the top of Fig. 15 (**Chirp**) is a simulated price curve that performs a regular oscillation with an increasing period from 10 to 60 bars per cycle. No very realistic pricing, but you can see what different spectral filters do with this curve. The bandpass filter (**BandPass**, second curve from the top) lets only a certain frequency range in the middle pass through. The high-pass filter (**HighPass**) dampens the trend and all longer-term cycles. The Laguerre low-pass filter behaves the opposite way, it attenuates all short-term cycles. At the very bottom for comparison you can see the traditional **EMA** indicator, a low-pass filter of the first order with significantly less effective attenuation.

What does a Laguerre lowpass filter do to a real price curve? In the image below, the grey line is the original EUR/USD price and the black line is the result from the Laguerre function:

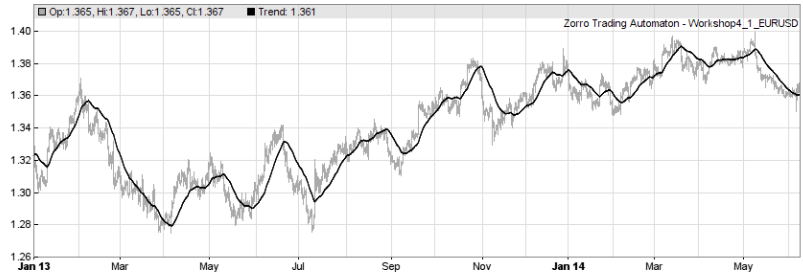


Fig. 16 – EUR/USD curve with a Laguerre lowpass

We can see that the lowpass filtered prices give a fairly smooth curve - the jags are removed. We can also see that the dark curve is slightly shifted to the right and follows the prices at a distance of a few days. So even a Laguerre lowpass filter produces some lag. For even shorter price movements, the trade signal would come too late. We will see the consequences in short.

Buying and selling

The next line in the script places a stop loss limit:

```
Stop = 10*ATR(100);
```

Stop is a predefined variable that Zorro 'knows' already, so we don't have to define it. It determines the maximum allowed loss of the trade. The position

is sold immediately when the price moves in the wrong direction by more than the given value. The limit here is given by **10*ATR(100)**. The **ATR** function is a standard indicator. It returns the Average True Range - meaning the average height of a candle - within a certain number of bars, here the last 100 bars. It is a weighted average, the last candles have more weight than the first. The position is sold when the loss exceeds the size of ten average candles. By setting **Stop** not at a fixed value, but at a value dependent on the fluctuation of the price, Alice adapts the stop loss to the market situation. When the price fluctuates a lot, higher losses are allowed. Otherwise trades would be stopped out too early.

A stop loss should normally be used in trade strategies. Not necessarily for limiting losses, but also for allowing the trade engine to better calculate the risk per trade and generate a more accurate performance analysis.

The line

```
MaxLong = MaxShort = -1;
```

limits the number of open trades to 1. The predefined variables **MaxLong** and **MaxShort** establish, as you might have guessed, a limit to the number of open long and short trades; a negative limit prevents a special treatment of open trades. Alice only wants one position open at a time for limiting risk. **X = Y = Z;** is C shorthand for **Y = Z; X = Z;** and makes use of the fact that the result of a C assignment can be assigned to another variable.

The following lines are the core of Alice's strategy:

```
if(valley(Trends))
  enterLong();
else if(peak(Trends))
  enterShort();
```

The **valley** function is a **boolean** function; it returns either **false** (**0**) or **true** (a value different to **0**)[1]. Here it returns **true** when the series just had a downwards peak. The **peak** function returns **true** when the series just had an upwards peak. The **if(..)** condition is then fullfilled, and a long or short trade with the selected asset is entered with a **enterLong** or **enterShort** command. If a trade was already open in the opposite direction, it is automatically closed by this command. And if the **MaxLong/MaxShort** limit would be exceeded, the command would not open a new trade. Note how we combined the **else** of the first **if** with a second **if**; the second **if** statement is only executed when the first one was not.

Let's check how such code could typically trigger a trade:

Fig. 17 – Trend change with a Laguerre filter

The dark line in the chart above is the **Trends** series - the lowpass filtered price. The peak on January 5 produces a **true** return value of the **peak** function, which in turn triggers a short trade. A trade is symbolized in the figure by a straight line connecting two small circles. This short trade was closed again on January 12 by opening a long trade. The latter due to a valley of the

[1] In fact a number is returned, since in lite-C **false** is the number **0** and **true** is the number **1**. Any integer result that is not **0** is interpreted as **true**.

trend curve and a subsequent **true** return value of the **valley** function. We see that the peaks and valleys of the trend curve lag about two days behind the price curve. If this were not the case, the short trade would already be open on January 4 and closed on January 10 (after the weekend, which is not visible in the chart). He would then have produced almost 5 cents, or 500 pips, of profit. Due to the lag, it was only about 80 pips. The delay of the trades by the lagging of signal curves has an enormous influence on the result of a strategy. Unfortunately, it can not be eliminated altogether – for this one had be able to look into the future. A normal moving average (SMA indicator), by the way, had with the same parameters not two days, but about two weeks lag on the price curve.

Now it's time to find out if Alice's strategy is profitable at all.

Testing a strategy

Start up Zorro. Select the script **Alice1a** (make sure before that you've downloaded the book scripts from **https://financial-hacker.com**). Make also sure that the asset is **EUR/USD** and the **Period** slider is at 60 minutes. Click **[Test]**:

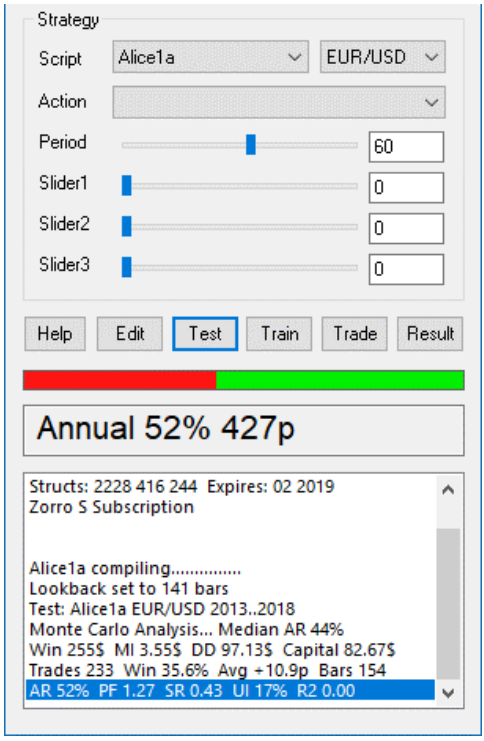


Fig. 18 – Test result

Zorro's default settings for a test are the last 6 years. For consistent results, all scripts of this book are set up for the test period 2013-2018. Still, you might get slightly different results since Zorro simulates trading with the current spread, leverage, and commission parameters of a default account – a micro lot account by a large broker - which usually change from release to release. So the test result can be better or worse dependent on current trading costs. To test different time periods, simply modify or remove the lines with **StartDate** and **EndDate** in the scripts.

By clicking on **[Test]**, the strategy performs a simulation with a price curve from the past - a procedure called 'backtest'. The strategy behaves as if it were traded during this period in fast-forward mode. So the 6 years run through in a few seconds. Zorro uses historical EUR/USD price data, which is available in the **History** folder. You can add other historical data to test the same strategy with other assets (and you'll find that this simple trend following strategy won't work with all assets).

The strategy achieved approximately 50% annual return on capital during the test period, equivalent to an annual profit of about 400 pips. To calculate the return, the average annual result is divided by the capital required for the maximum margin of open trades and for covering the maximum loss that can be expected in a 3 years period. At first glance, 50% is not a bad result, at least in comparison to the lousy 1% of a savings account. Still, Alice is not too excited. The reason can be seen in the profit curve. Click **[Result]**:

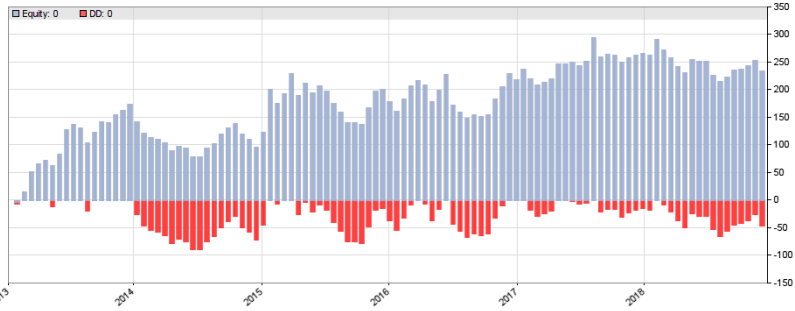


Fig. 19 – Equity curve

What you now will see depends on your Zorro version. With Zorro S, an interactive chart viewer will pop up with which you can zoom in or step into the chart that results from the test. With the free Zorro version, the chart is just a fixed resolution image. In any case it should look as in Fig. 19.

The upper blue area is the **equity**, the accumulated profits of all open and closed trades minus their losses. The red 'underwater' curve below the zero line is its evil counterpart, the **drawdown** curve that indicates losses on our account. And we can see the problem right away. The equity increases from the beginning to the end, but by no means uniformly. It rises nicely only in a few years. In those years the main profit of the whole test was generated. But if Bob had started in 2014 with Alice's strategy, he would had accumulated an ugly loss by the end of the year. Most traders would probably have given up by then and demanded the money back from Alice.

It is not enough for a strategy to make good profits. The profit should also be as uniform as possible, without long and deep loss phases in between. A savings account has no drawdown - at least as long as the bank does not go bankrupt. For this reason, traders have developed a measure to compare the return on a strategy with the return on a savings account: the **Sharpe Ratio**.

The Sharpe Ratio is the profit in relation to the risk[1]. The measure of risk is the fluctuation in profit. A savings account with 1% interest would have a Sharpe ratio of 1.01. A Sharpe ratio above 1 means fairly safe gains. If it is below 1, but above 0, the system is still profitable, but the risk is higher than the profit. Systems that make loss have a negative Sharpe ratio. The Sharpe ratio of Alice's system - visible in the message window as 'SR' - is about 0.4. Not actually a great value.

In the message window we can see many more results and parameters of the test. But before we turn to them, let's look at the trades that Alice's strategy has made. Perhaps we find here the secret of the low Sharpe ratio. Here's again a result chart, this time with the trades:

Fig. 20 – Simulated trades

Each trade is represented here by a green or red line in Fig. 20. The lines connect the start and end points of the trades, shown in the chart as small circles. Green (light) is a winning, red (dark) a losing trade. You can see immediately that there are far more red than green points – most trades are lost. However, the longer lasting trades have all the green lines. So we have many small losses (red dots) and few big gains (green dots and lines). This is typical of a trend following strategy.

[1] Formula: Average gain divided by its standard deviation. The Sharpe ratio is calculated here without deduction of a risk-free return, since the latter would make little sense for comparing trade systems.

Analyzing the trades

Since Alice now wants to check the single trades of this strategy in detail, she adds the following line to the **run** function:

```
set(LOGFILE);
```

LOGFILE is a **flag** - something like a "switch" that can be on or off. Such switches are turned on with the **set** function. If the switch is on, the next click on **[Test]** stores a protocol of all events in the **Log** subfolder. Add the line, click **[Test]**, then open **Log\Alice1a_test.log** with the script editor. It is a long list with entries that look like this:

```
[8372: Fri 21.05.14 05:00]  -1220p 29/141
[EUR/USD::S4070] Reverse 1@1.2631: +60.23 at 05:00
[EUR/USD::L7271] Long 1@1.2631 Risk 13$  at 05:00

[8373: Fri 21.05.14 06:00]  -381p 29/142
[8374: Fri 21.05.14 07:00]  -381p 29/142
[8375: Fri 21.05.14 08:00]  -381p 29/142
[EUR/USD::L7271] Reverse 1@1.2503: -9.35 at 08:00
[EUR/USD::S7572] Short 1@1.2503 Risk 14$  at 08:00
```

If nothing happens during a bar, only the bar number (**8372**), the time (**Fri 21.05.14 05:00**), the current profit or loss in pips (**-1220p**) and the numbers of won (29) and lost trades (141) so far are recorded. What can happen is closing or opening a position. Here, the short position **S4070** is first closed by opening a long position in the reverse direction. The other option of the script to close a position would be a stop loss that would be indicated by **Stop**. 1 Lot was sold for the price of 1.22631 USD (**1@1.2631**). This resulted in a profit of $ **60.23**.

The long trade that has closed the previous short position is opened in the following line. He got the number **L7271** assigned; the 'S' or 'L' in the number stands for 'Long' or 'Short'. 1 Lot EUR/USD is purchased at the price of **1.22631** USD. The risk - the maximum loss of this trade when the stop is hit - is $ 13 (the **$** means that the number indicates a money amount, a postponed **p** would mean pips). The 13 dollars is an estimate based on commission, spread, and the distance of the stop loss from the current price. The actual loss at a stop can be somewhat higher due to slippage.

In fact, the trade loses. It does not trigger the stop, but is closed 3 hours later by a trade in the opposite direction with a minus of 9.35 in account currency

units. You can see in the log that most trades are lost. Zorro seems to deliberately enter trades in the wrong direction; trading at random would only lose a little more than 50%, not 65%. Surely no human trader in his right mind would enter trades this way! But there's a method behind this madness. The algorithm wants to be in a favorable position when a long-term trend begins, and then keep the position for a long time. That's why it wins in the long run despite losing most trades.

For some more insight into the strategy behavior, Alice wants to plot the profit distribution. For this she adds the following line to the very begin of the script (before the run function):

```
#include <profile.c>
```

This is a command to the compiler to insert another script file from the **include** folder. **profile.c** is a script that contains functions for plotting price and trade statistics and seasonal analysis charts. Due to the #include line these functions are now available to the script. For the profit distribution, Alice calls the following function from inside the run function:

```
plotTradeProfile(-50);
```

Add this call at the end of the **run** function. Clicking **[Test]** (don't forget to save the script before) will now produce a graphical representation of the profits and losses in steps of 50 pips. The generated histogram looks like this:

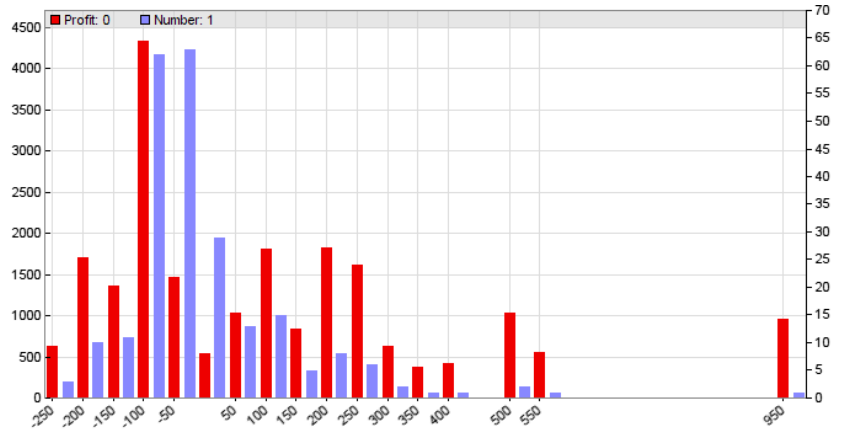


Fig. 21 – Histogram of wins and losses

For generating the histogram, all trades are sorted into buckets, depending on their profit. Every bucket is represented by a red and a blue bar (black and grey in the image). Trades with a loss between -250..-200 pips go into the first bucket at the left side, marked -250 at the x axis. The next buckets are for trades with loss or profit from -150..-100 pips, -100..-50 pips, and so on. The height of the grey bar is the number of trades ending up in that bucket (right y axis), the height of the black bar is the sum of all profits or losses in the bucket (left y axis). We can see that most trades end with a loss between 0 and -50 pips. The total profit of the system comes from relatively few profitable trades, some even with almost 1000 pips profit.

This is a suboptimal profit distribution. With only few winning trades, the system is a bit too dependent on chance. It also requires strong nerves to trade it even automatically, as it will most likely begin with a loss streak and an equity drawdown. And it suffers anyway from drawdowns all the time. Can Alice come up with some idea to improve the system a little?

The Market Meanness Index

All trend following systems have the same problem: the market shows only temporarily strong trends. At other times, the price is moving sidewards without a clear direction up or down. A trend following system will then generate false signals and accumulate losses. This problem can be seen in Fig. 19 and Fig. 20, where a lot of red (dark) dots cluster during a sideways movement of the price. They also generate strong dents in the equity curve. So it would be good to detect such situations and to prevent or at least reduce trades during this time.

There are various algorithms and indicators to distinguish a developing 'trendy' market situation as early as possible from sideways movement. Some of them actually work, for example the Hilbert transform used by John Ehlers or the Hurst exponent used by Benoit Mandelbrot. Alice wants to use a statistical algorithm, which is available as an indicator: the **Market Meanness Index** (**MMI**). You can find its source code and a description of the algorithm in Appendix 1 of the book. Fot the moment it shall be sufficient that the MMI is based on the statistical distribution of price data. Its returned value is between 0 and 100%. It shows how strong the market is fighting against the trend. At 100%, any price movement would be compensated immediately by a counter-movement, and at 0% a trend would continue indefinitely. Pure random numbers have an MMI of 75%. For price data, which are 'trendier' than random data, the MMI is normally much lower. The MMI

going down is a hint of a just developing trend. A falling MMI is therefore favorable for trend following systems, an increasing MMI is unfavorable.

To detect the market situation, Alice first calculates the MMI over the last 300 hours (Alice1b script):

```
vars MMI_Raws = series(MMI(Prices,300));
```

The result is a series of 'raw' MMI values. In order to make a smooth indicator from these fluctuating data, Alice calculates its average:

```
vars MMI_Avgs = series(SMA(MMI_Raws,300));
```

Fig. 22 – MMI, raw and smooth

The **S**imple **M**oving **A**verage, implemented in the **SMA** function, sums up the last 300 MMI values and divides the sum by 300. This results in the average MMI of the last 300 hours. In Fig. 22 you can see the MMI curve - gray the raw values, black the averages - in the window under the price curve. The MMI ranges between 50% and 60%. At first glance, the MMI curve, raw or averages, does not show any similarity with the price curve. The significance becomes clear only when the smoothed MMI is used as an additional trade condition:

```
if(falling(MMI_Avgs)) {
  if(valley(Trends))
    enterLong();
  else if(peak(Trends))
    enterShort();
}
```

It is now only possible to open a position if the MMI falls and thus announces a trendy market situation. Let us look at how this modification affects the outcome (**Alice1b**, **[Test]**):

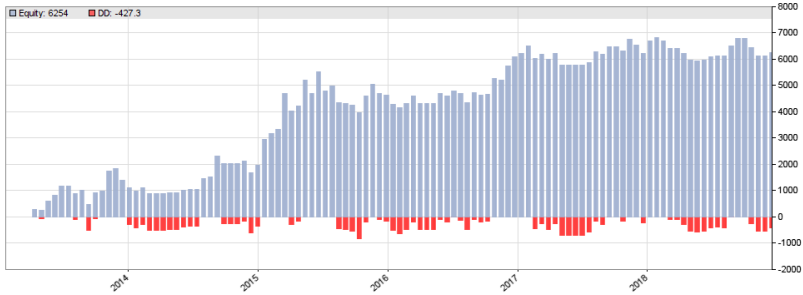


Fig. 23 – Result with MMI

We have significantly increased the profit, and the equity curve looks better. The loss at the beginning is neutralized, and the equity slump in 2014 is gone. Although the MMI has also filtered away a few previously profitable trades, the benefits outweigh the damage. As the efficiency of the market often fluctuates, strategies should always have filter algorithms that suppress trades when the underlying inefficiency is not present. This crucial aspect is often forgotten in trading systems. Especially for trend following systems, recognizing profitable and unprofitable market situations is almost more important than generating the trade signals themselves.

The average monthly income (**MI** in Zorro's message window) is 4 dollars. Pretty modest, but our position size is only 1 lot and Zorro's simulation is based on a microlot account by default. With this setting, Zorro requires only about 60 $ deposit (**Capital** in the window above - the $ sign means not necessarily US dollars, but the currency of your broker account) to run the stratey in a rather safe distance from a margin call. So you have about 5% return on investment per month.

Performance measurement

Some of the values in the Zorro window are directly derived from the simulation, others - like the **Sharpe Ratio**, which was already briefly mentioned - require more or less complex calculations. All those number only serve to answer two questions: How profitable is the strategy? And how likely is it to

get the same result in the actual trading? Both aspects together are the performance of a strategy. The first question is easy, the second and more important one, unfortunately, difficult to answer.

For this reason, a strategy is analyzed under many different aspects, so that one gets a mosaic-like impression of their behavior. The result is a series of key figures, the most important of which are displayed in the Zorro window. For more details, click **[Result]** and a performance report will appear in the script editor window. Such a report begins like this:

```
Test Alice1b EUR/USD

Simulated account   AssetsFix
Bar period          1 hour (avg 86 min)
Test period         2013-01-21..2018-12-31 (35970 bars)
Lookback period     300 bars (18 days)
Montecarlo cycles   200
Simulation mode     Realistic (slippage 5.0 sec)
Avg bar             16.8 pips range
Spread              1.5 pips (roll -0.10/-0.11)
Contracts per lot   1000.0
```

If nothing else is set up, the simulation starts in January six years ago and ends with the current date or with the end of the historical price data. The actual test period starts after the lookback period, which is the part of the price curve with which the used indicators and functions must be fed until they can deliver the first results. When using a moving average of 300 bars, the simulation can begin to trade at the earliest from bar 300 on. The necessary lookback time is normally automatically calculated by Zorro. Otherwise it can be set up with the **LookBack** variable.

In the test a slippage of five seconds was assumed. This does not mean that any order is delayed by five seconds, but that the asset is bought or sold at a price randomly taken from a range of five seconds after the current price of the price curve. This price can be worse or better than the current price. Most of the time, as you might expect, it is worse.

The next values in the report provide information about the simulated asset, here EUR/USD. They usually come from the current account data. The bid/ask spread is currently 0.3 pips, the rollover 10 cents loss per day for long trades and 11 cents loss for short trades, per 10,000 units. 1 Lot are 1000

units since we simulate a Mikrolot account. All those asset specific parameters are loaded from the broker and set up in a spreadsheet ("AssetsFix.csv") in the **History** folder.

```
Gross win/loss      707$ / -408$ (+2983p)
Average profit      50.19$/year, 4.18$/month, 0.19$/day
Max drawdown        -63.94$ 21.4% (MAE -88.92$ 29.8%)
Total down time     69% (TAE 79%)
Max down time       63 weeks from Jan 2017
Max open margin     13.80$
Max open risk       33.03$
Trade volume        98633$ (16596$/year)
Transaction costs   -12.30$ spr, -0.24$ slp, -18.26$ rol
Capital required    59.22$
```

The gains and losses accumulated at the end of the simulation are displayed as **Gross win/loss** in the report; the difference appears under **Profit** in the window and as pips in the report (a trailing **$** indicates an amount in the account currency, a small **p** stands for pip). The monthly profit - total profit divided by the number of months in the test period - is given as **MI** (**M**onthly **I**ncome) in the window.

DD in the window is the maximum drawdown, the largest capital loss during the simulation. It is calculated form the difference between the highest balance peak and the next deepest equity valley[1] and is an important value for determining profitability. The smaller the drawdown, the less capital is needed to achieve the same result. The report also shows the drawdown in percent of profit as well as the maximum decline in equity (**MAE**, **M**aximum **A**dverse **E**xcursion). This is usually slightly higher than the maximum drawdown, as temporary intermediate gains of open trades are included in it. A

[1] Many trading programs calculate two different drawdown values, one from the balance curve and one from the equity curve. The balance drawdown has no practical use as far as I see. The equity drawdown is a more useful metric. But the really relevant drawdown that determines the actual capital requirement must be calculated from both curves – it's the difference of the highest balance to the subsequent lowest equity.

big difference between drawdown and MAE could indicate that trades are closed too late and thus often give away their initial profit.

Down time is the time during which the strategy loses and the capital drops. It is often above 50%, especially for trend following systems. Bob will be in a bad mood here in 69% of all cases when looking on his account. The longest drawdown of this strategy lasted 63 weeks, more than a year.

In addition to the maximum drawdown, the **Largest margin** is also important for calculating the capital requirements of a strategy. For forex trading systems, margin is often negligible compared to the drawdown. But for some strategies, such as grid traders, or for stock tradings systems it can take high values. **Trade volume** is the total value of all traded assets. The transaction costs are distributed over spread (**Spr**), slippage (**Slp**), rollover (**Rol**) and commission (**Com**). Slippage and rollover can also in rare cases also take positive values and then increase the profit instead of reducing it.

The **Required capital** is the sum of the maximum drawdown and the maximum margin, ie the money requirement for open trades. This capital would be needed to survive the simulation period when it was entered at the worst possible time, immediately before the largest drawdown with the maximum volume of open trades. In order to make the capital requirement independent of the simulation time, the drawdown is previously converted to a fixed period of three years. As we shall see later, longer simulation periods produce higher maximum values for the drawdown[1].

```
Number of trades    82 (14/year, 0/week, 0/day)
Percent winning     43.9%
Max win/loss        89.41$ / -20.79$
Avg trade profit    3.64$ 36.4p (+196.3p / -88.7p)
Avg trade slippage  -0.0028759$ -0.0p (+1.5p / -1.2p)
```

[1] With a balanced strategy that neither wins nor loses on average, the maximum drawdown increases over time proportionally to the square root of the number of trades.

```
Avg trade bars      355 (+662 / -115)
Max trade bars      1888 (16 weeks)
Time in market      81%
Max open trades     1
Max loss streak     6 (uncorrelated 8)
```

The next section in the report provides information about the number of trades, the percentage of trades won and the highest profit and loss. Average profit and slippage per trade are given as money amount, in pips as well as separately for won (+) and lost trades (-). The average and maximum duration of a trade (**Avg / Max TradeBars**), the time when trades are open (**Time in market**) and the number of simultaneously open trades (**Max open trades**) give indications of the strategy's exposure to the uncertainties of the market. **Max Loss Streak** is the highest number of successively lost trades, in the simulation as well as in theory, assuming independent random events (**uncorrelated**). So you have to prepare for these losses when the strategy is actually being traded. Strong deviations between simulated and theoretical value indicate that the results of individual trades are not completely independent. A loss is then followed by a loss with a slightly higher probability, and a profit to the next profit. This can be taken advantage of, as we shall see later in Chapter 6.

```
Annual return       85%
Profit factor       1.73 (PRR 1.26)
Sharpe ratio        0.55
Kelly criterion     0.36
R2 coefficient      0.680
Ulcer index         12.2%
```

The last section contains the key figures. If one knows the profit and the necessary capital, the annual return can be calculated. It is simply the percentage of the annual gain divided by the required capital, and is displayed as **AR** (**A**nnual **R**eturn) in the message window. These are the 85% from the last variant of Alice's strategy. Of course, this is only the average value of all years - and most importantly, it refers to the simulation! You can not rely on this return in real trading. The probability of this is indicated by some of the other key figures.

The **Profit factor** (**PF** in the message window) is simply the sum of all profits divided by the sum of all losses. It is a traditional measure for comparing strategies. A neutral strategy that neither wins nor loses in the long term has a profit factor of 1. For most profitable strategies, the profit factor is in the range of 1.1 to 1.8. The **P**essimistic **R**eturn **R**atio (**PRR**) is the profit factor

reduced by a penalty value for systems that perform very little trades and whose key figures are therefore more uncertain.

The **Sharpe Ratio** (**SR** in the window) takes into account the risk associated with the return. It is the quotient of the annual profit and its standard deviation. The more the profit fluctuates, the higher - so it is assumed - is the risk and the lower the Sharpe Ratio.

The **R2 coefficient** measures the linearity of the equity curve. It is compared with a line through its start and end points. The more the curve approaches a straight line, the higher the R2 coefficient and the more promising the strategy. With a coefficient of 1 - which in practice is never achieved - the equity curve would be a straight line and the profits would be streaming in continuously. In the Alice2a script, however, the R2 coefficient is 0, thus indicating a bad equity curve. And in fact, the strategy makes its profits mainly in a few years, otherwise the equity fluctuates.

The **Ulcer Index** (**UI**) takes its name from the stomach ulcers causeb by long and deep drawdown periods. It indicates the average depth of a drawdown in percent. The higher the ulcer index, the stronger a stomach you need to trade thise strategy. Good strategies should have an ulcer index below 10%.

All these values are only calculated from the simulation of the strategy with a very specific historical price curve. But what happens if the market conditions are the same, but the price curve of the asset happens to take a different course?

The Monte Carlo method

To answer this question, Zorro runs an additional analysis using the Monte Carlo method. Compared to the normal simulation, the Monte Carlo method produces more accurate and less random results.

In the Monte Carlo analysis, the equity curve from the simulation is split into many individual sections and these are replaced with one another in random order. This creates many new equity curves, each representing a different order of the trades and different price movements within the trades. All the curves have the same start and end points, so the total profit stays the same, but the way to reach it is different any time, and therefore also the drawdown, the required capital, and the annual return. The curves are then evaluated and sorted by their annual return. Finally, a confidence interval is assigned to each

curve, which indicates in percent how many of all curves yield the same or a better return.

Without Monte Carlo analysis, the annual return would only be calculated from the single equity curve of the simulation. Let's say it's 100%. The Monte Carlo analysis now calculates the yields of hundreds or thousands of different curves and assigns the results to their confidence intervals. In our example, a return of 100% could then correspond to a confidence interval of 60%. This means that 60% of all analyzed curves yield annual returns of 100% or more. It also means that in 40% of all cases the return was worse.

The Monte Carlo analysis is particularly helpful in estimating the risk and capital requirements of a strategy. The maximum drawdown in the simulation is normally used as a measure of the risk. This normally means that our risk assessment is based on a historical price curve, which is unlikely to ever repeat. Even if the market conditions and the statistical distribution of the trades in the simulation were the same as in the real trades, the order of the results is largely a matter of chance. With a series of several losses in a row, you can get a very large drawdown; but with a different order of trades so that the losses spread more evenly, the maximum drawdown is much smaller. This randomness in the risk assessment can be eliminated by Monte Carlo analysis by generating many different equity curves and thus many different sequences of trades.

Here is the distribution of annual returns of Alice's trend following strategy in the results report:

Confidence level	AR	DDMax	Capital
10%	74%	76	67.70$
20%	67%	86	75.11$
30%	61%	97	82.82$
40%	56%	106	88.86$
50%	54%	112	93.61$
60%	51%	120	99.32$
70%	46%	135	110$
80%	41%	152	122$
90%	35%	185	145$
95%	29%	221	171$
100%	16%	410	305$

In 10% of all cases we achieve a return of 74% or better, while in 95% of all cases the return is 39% or better. The above annual return of 85% from the

simulation was therefore a lucky outlier: it would only be achieved in maybe 5% of all cases. Zorro displays in the message window also the median AR, which is the minimum return in 50% of all cases. Some traders hope to be on the safe side when they only trust results in the 95% interval, and expect only 29% return. Each trust interval also has a different maximum drawdown (**DDMax**) and a different capital requirement (**Capital**). The larger the interval, the lower the expected rate of return and the higher the capital required to trade. For Zorro, the desired interval can be determined with a variable (**Confidence**, see Zorro manual).

However: even the results of a Monte Carlo analysis in a 95% confidence interval are by no means guaranteed in real trading! If market conditions have changed drastically, or if the results of the simulation have been distorted by wrong test methods, something quite different can emerge. How to minimize such nasty surprises is one of the topics in the next chapter.

6 tricks for trend trading

► For detecting trend, use functions with low lag and fast reaction. The classical ME or EMA is not really suited. Better use a Laguerre filter or other **lowpass filters** that smoothen a price curve without much delaying the signal.

► The usual crossover also produces lag. Better use functions like **valley** or **peak** that trigger buy and sell signals directly after trend changes.

► Use an **adaptive stop loss**. If you don't need one, place at least a very distant stop.

► Always check out single trades in the **log file**. Also look into the profit/loss distribution. Both give valuable hints about improving the system.

► Any trend following system needs a filter for recognizing unprofitable market situations. The **Market Meanness Index** or similar filters prevent trading in such situations and can significantly improve the profit.

► Get familiar with all details of the **performance report**. It contains key figures for the profitability of a strategy and also a **Monte Carlo analysis** that method removes randomness from the results.

4 Against the trend

Bob: "Last week I ran into Warren Buffett. I naturally asked him for a trading advice. That's what he said: 'Be greedy when others are fearful'.
Alice: "Interesting. And what does that mean?"
Bob: "He didn't tell, but went away. But I think he wanted me to trade against the trend. "
Alice: "Is that not the exact opposite of the last strategy?"
Bob: " You got it. But it is logical: Buy cheap and sell dear. I will now buy long when prices moved down a lot and go short when they moved up a lot. I need you to automatize this."
Alice: "How much is very much?"
Bob: "Depends on the market."
Alice: "I should have known."
Bob: "Well, prices often go up and down in cycles. Just find out if there is such a cycle. If then the price falls, we know that it will soon rise again. So we buy long as soon as the price goes below a certain point in the cycle. With short trades, we do the opposite."
Alice: "Hm. I could use a bandpass filter to remove the slow trend and the fast noise from the price curve. Then only the price cycles remain. "
Bob: "Sounds good."
Alice: "Then I normalize the curve so that the cycles get the same average amplitude. If the curve then rises above a certain threshold, I go short, below a threshold I buy long. Is this ok?"
Bob: "Still sounds good."
Alice: "Before the comparison with the threshold, I apply a Fisher transformation. This gives the curve a Gaussian distribution with sharp and well defined oscillations, so we get less false signals."
Bob: "I have no idea what you're talking about. But sharp and well defined sounds good. "
Alice: "Of course, it's more complicated than a trend following system. It will be a little bit more expensive. "
Bob: "That sounds less good."
Alice: "But I can also do a parameter optimization for the price. And above all, an out-of-sample test."
Bob: "Does this bring more profit?"

Alice: "Not necessarily. But the profit can be better predicted. I want to be sure that you can afford me."

The cycle trading algorithm

This is the first version of Alice's counter trend trading script (**Alice2a** script; agreed fee: $2000):

```
function run()
{
  BarPeriod = 240;

  vars Price = series(price());
  vars Cycles = series(BandPass(Prices,30,2));
  vars Signals = series(FisherN(Cycles,500));
  var Threshold = 1.0;

  LifeTime = 100;
  Trail = Stop = 10*ATR(100);
  MaxLong = MaxShort = -1;

  if(crossUnder(Signal,-Threshold))
    enterLong();
  else if(crossOver(Signal,Threshold))
    enterShort();

  plot("Cycles",Cycles,NEW,RED);
  plot("Signals",Signals,NEW,GREEN);
  plot("Threshold+",Threshold,0,BLACK);
  plot("Threshold-",-Threshold,0,BLACK);
  set(LOGFILE+PLOTNOW);
}
```

Counter trend trading is affected by market cycles and more sensitive to the bar period than trend trading. Bob has told Alice that bar periods that are in sync with the worldwide markets - such as 4 or 8 hours - are especially profitable with this type of trading. Therefore, she has set the bar period to a fixed value of 4 hours, or 240 minutes:

```
BarPeriod = 240;
```

BarPeriod is one of the predefined Zorro variables, which can be used to determine the behavior of the strategy and the analysis. These variables are all described in the Zorro manual (click **[Help]** in the Zorro window and open the "System Variables" section).

The counter trend trade rules are contained in the following lines that calculate the buy/sell signal. The first line sets up a price series just as in the trend trading strategy:

```
vars Prices = series(price());
```

In the next line, a bandpass filter is fed with the price curve:

```
vars Cycles = series(BandPass(Prices,30,2));
```

The behavior of a bandpass filter can be seen in the second curve of Fig. 15 in the previous chapter. Its function is similar to a lowpass filter except that it dampens not only low but also high frequencies, i.e. short cycles. This bandpass filter has a center period of 30 bars and a width of 2 (the width determines the frequency range that can pass the filter). This way the trend (a cycle with a very long period) and the noise (short period cycles) are removed from the price curve. The result is a clean curve that consists mostly of the peaks and valleys of the 30-bars cycle, equivalent to 1 week (30*4 hours = 5 working days). It's stored in a new series named **Cycles**.

Just like the original prices, the values of the **Cycles** price curve are still all over the place. For generating a trade signal, they must be normalized - meaning they are 'compressed' in a defined range so that they can be compared with a threshold. In traditional technical analysis, an indicator called "Stochastic" is often used for normalizing a curve. Alice prefers the **Fisher Transformation**. This is an operation[1] that transforms a curve into a **Gaussian distribution** - that's the famous 'bell curve' distribution where most val-

[1] Formula: $f(x) = \frac{1}{2}\log(\frac{1+x}{1-x})$

ues are in the center and only few values are outside the +1...-1 range. Normalization and Fisher transformation are done with the **FisherN** function. It converts the **Cycles** series into the normalized and Gaussian distributed **Signals** series, using the last 500 bars for the normalization.

```
vars Signals = series(FisherN(Cycles,500));
```

The **Signal** series can now finally be compared with an upper and lower threshold for generating trade signals. The Threshold is defined in the next line:

```
var Threshold = 1.0;
```

This line defines a new variable **Threshold** with a value of 1.0. Alice's intention is to let any **Signals** value that leaves the +1.0 ... -1.0 range trigger a trade. This happens in the following part of the code. But before we can start trading, Alice limits the trade duration:

```
LifeTime = 100;
```

Since we're trading against the trend, trades should not stay open indefinitely. **LifeTime** automatically closes any trade after 100 bars, it it wasn't closed before.

```
Trail = Stop = 10*ATR(100);
```

This line places a trailing stop loss at an adaptive distance from the price, just as in the trend trading script. The **ATR** function is again used to determine the stop loss distance at the average height of 10 candles. Additionally to the stop loss, Alice has also placed a trail trigger at the same distance. **Trail** moves the stop loss when the trade becomes profitable. If the trade goes in favorable direction by more than 10 average candles, the stop loss will follow the price at a distance of 20 average candles (original stop distance plus trail distance). This ensures that all trades that reach a 20 candles profit are guaranteed to end with a win, regardless how the price further behaves. Trailing often - not always - improves the profit of a strategy and is almost always better than placing a profit target.

```
if(crossUnder(Signals,-Threshold))
  enterLong();
else if(crossOver(Signals,Threshold))
  enterShort();
```

The **crossUnder** and **crossOver** functions work similarly to peak and valley. You can test whether two curves intersect each other or whether a curve crosses above or below a threshold value. As soon as the **Signal** curve crosses the negative threshold value (**-Threshold**) from above – meaning when **Signal** falls below -1 - a long trade is triggered by **reverseLong**. Since, in the case of a Gaussian distribution, only a few values are below -1, the price should be close to a low point of the oscillation when this threshold is reached, and should consequently soon rise again. If, conversely, the positive threshold value is crossed from below, the price is supposed to be close to a maximum and triggers a short trade. This system therefore behaves in the opposite way as the trend following strategy. Therefore, one might expect to make profit with this system when trend following fails, and vice versa.

Plotting signals

Obviously, the counter trend trade rules are somewhat more complicated than the simple lowpass function of the previous chapter. So Alice needs to see how the various series look like, for checking if everything works as supposed. This happens in the last lines at the end of the script:

```
plot("Cycles",Cycles,NEW,RED);
```

This line generates a plot of the **Cycles** series. It's plotted in a **NEW** chart window with color **RED**. We can use the **plot** function to plot anything into the chart, either in the main chart with the price and equity curve, or below the main chart in sub-charts.

The **Signals** curve and the upper and lower **Threshold** are plotted in another sub-chart window:

```
plot("Signals",Signals,NEW,GREEN);
plot("Threshold+",Threshold,0,BLACK);
plot("Threshold-",-Threshold,0,BLACK);
```

The first statement plots the Signal series as a green curve. The next two statements plot the positive and negative **Threshold** with two black lines in the same chart window.

```
set(LOGFILE+PLOTNOW);
```

PLOTNOW is a flag that lets the chart pop up after the test, without clicking **[Result]**. Load the script **Alice2a** and make sure that **EUR/USD** is selected, then click **[Test]**:

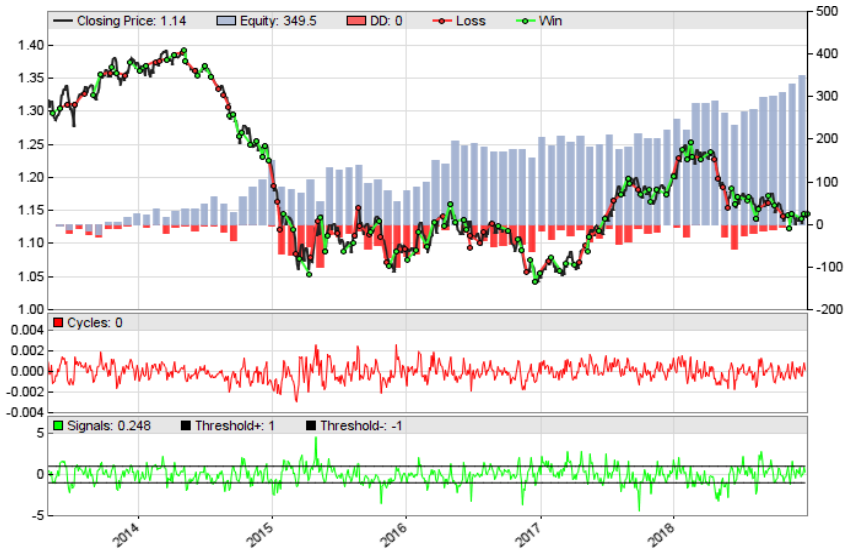


Fig. 24 – Counter trend strategy

The curve in the middle chart is the plot of the **Cycles** series. It shows the price fluctuation in the range of about +/-0.002, equivalent to about 20 pips. The bottom chart displays the **Signals** series. The black lines are the thresholds that trigger buy and sell signals when **Signals** crosses over or under them.

Plotting variables and series in the chart greatly helps to understand and improve the trade rules. For examining a trade in detail, click on its position on the Zorro S chart and zoom into it with the mouse wheel or with the **[Zoom+]** button:

Fig. 25 – Chart detail

This is also possible with the free Zorro version, but a bit less convenient. Set up a start date and chart length with the predefined **PlotDate** and **PlotBars** variables, then run the test again. The selected part of the chart will now be displayed.

We can see that the script generates a positive return, although we again have some profit bursts followed by long unprofitable periods. But we also note something interesting: The red **Cycles** curve got smaller fluctuations in 2014 and 2017, indicating a lower volatility of the EUR/USD (the **Signals** curve does not show this effect because it's normalized). At the same time, the price curve was strongly trending. Obviously the market goes through different regimes, non trending with high volatility and trending with low volatility. Can the be used to filter out unprofitable market periods? After all, this had worked with trend trading.

Fractal dimensions

Alice has generated a cyclical trade signal with a bandpass filter from the price curve. This obviously works well when the curve has such a cycle, but less well when the prices behave differently. It would appear a good idea to use the MMI as a filter again, but with the opposite sign - not trading in trends,

but trading when the market does not show a trend. Unfortunately, this would not work. The MMI would interpret regular market cycles also as a non-random, trending curve.

Fortunately there are more indicators for detecting the market regime. The **FractalDimension** indicator measures the length of the path that the price follows when going from one point to another. If the price curve has many ripples and jaggies, its price path is long and its fractal dimension is high. If it goes in a straight line, its fractal dimension is low. This effect can be used to distinguish a cycling market regime from a trending one. The following diagram shows how the **FractalDimension** and the **MMI** indicators respond to a market change:

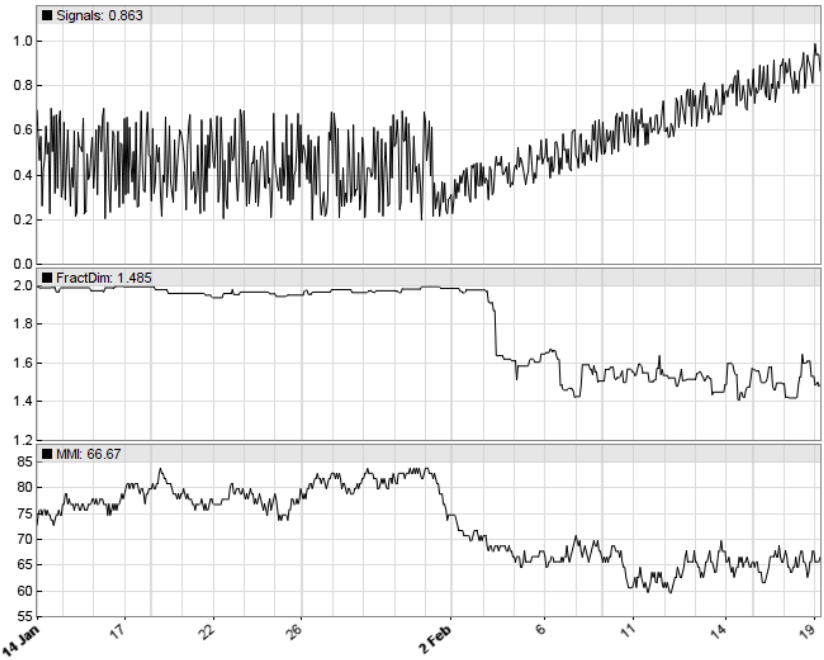


Fig. 26 – Detecting trend and market cycles

You can reproduce such diagrams with the **Regime** script. In the top chart we have simulated a transition from a sidewards market with significant cycles to an upwards trend with smaller cycles. The middle chart shows the fractal dimension. Heavy cycling 'smears' the price curve to a two-dimensional area, as opposed to a straight line that had only one dimension. We can see how the **FractalDimension** drops – with some delay – when the market begins trending and the volatility goes down. The bottom chart shows the

MMI response on the same situation. We can see that the fractal dimension does not catch the trending start very fast, but distinguishes the different market regimes more clearly.

Here's Alice's code to establish a trade filter with the **FractalDimension** indicator (**Alice2b** script):

```
var Regime = FractalDimension(Prices,100);
if(Regime > 1.5)
{
  if(crossUnder(Signals,-Threshold))
    enterLong();
  else if(crossOver(Signals,Threshold))
    enterShort();
}
```

Restricting the trading to a 1.5-dimensional price almost doubles the backtest result.

With such results, there's always the question whether the system can achieve the same performance in live trading. Or does the result come mainly from a lucky choice of parameters, which coincidentally just fit the historical price curve? To determine how robust a system is – how it responds to changing conditions in the real market - a simple backtest is not enough. Alice needs to train the strategy.

Training

Each strategy consists, on the one hand, of the **algorithm** – that is, the trading rules of the system - and, on the other hand, of a number of **parameters**, numeric values used by the algorithm. This could be the bandpass filter cycle (30 in the script above), the factor for the stop-loss distance (4), and other numbers that have a significant effect on the trade behavior. Training tests the behavior of a strategy when these parameters are changed.

Three objectives are to be achieved. The first one is to find out whether the profit of the strategy comes not from a random combination of parameters, but actually from the trade method. Only then can one expect that the strategy exploits a real inefficiency and will also be successful in real trading. The second objective is improving the 'robustness' of the strategy. The training process attempts to find the 'sweet spots' of the parameter values, where small value changes have little effect on the strategy. This makes the strategy

insensitive to modest changes in the market. The third objective is to automatically adapt the strategy to different markets and assets. The same strategy often works with many similar assets, but each asset needs different parameters. For this purpose, the training process is carried out separately for each asset, and produces an individual set of parameters.

Training differs in some aspects from the 'parameter optimization' provided by some trade platforms. Parameter optimization determines a value combination in which the strategy produces the highest profit with the historical price curve. Of course, this results in the best performance results in the backtest - but not in actual trading. On the contrary, the strategy with optimized parameters often fails miserably with slightly different market conditions and price curves. Optimization - especially with genetic algorithms - thus basically is the opposite of training. It produces not the most robust strategy, but the most adapted to the historical price curve. This effect is referred to as **overfitting** or **curve fitting**. If one imagines the profit of a strategy as a rugged surface in the parameter space, the optimization always finds the lonely peak of this surface. Since other price curves and market conditions always shift and distort this profit surface, the real trading can suddenly fall into a deep loss valley where the simulation with historical data had still the lonely profit peak. Then the strategy will burn your money despite great backtest values.

Curve fitting is one of the main problems when developing strategies. Some strategy developers therefore generally avoid the optimization of their parameters. However, this is not a real solution as the initial parameters selected manually can also be on a random profit peak. This happens more often than you think. Hardly anybody will resist the temptation to tamper with the parameters during development of the strategy, and to adjust them in such a way that the greatest possible profit comes out.

Zorro uses a different approach. The training is not looking for a profit peak, but for a rather broad profit plateau in the parameter space. This plateau might be at a distance from the highest peak, but still produce good profit. Which parameters should be adapted in which way is determined in the script. The parameter training is thus not a separate process, but an integral part of the strategy, which can be carried out at any time, even during live trading.

Alice has added some new commands to her strategy (script **Alice2c**):

```
function run()
{
```

```
  set(PARAMETERS+LOGFILE+TESTNOW+PLOTNOW);
  BarPeriod = 240;
  LookBack = 500;
  if(Train) Detrend = TRADES;

  vars Prices = series(price());
  vars Cycles = series(BandPass(Prices,30,2));
  vars Signals = series(FisherN(Cycles,500));
  var Threshold = optimize(1,0.8,1.2,0.1);

  LifeTime = optimize(100,50,150,10);
  Trail = Stop = optimize(10,4,20,2) * ATR(100);
  MaxLong = MaxShort = -1;

  var Regime = FractalDimension(Prices,100);
  var RegimeThreshold = optimize(1.5,1.3,1.7,0.1);
  if(Regime > RegimeThreshold) {
    if(crossUnder(Signals,-Threshold))
      enterLong();
    else if(crossOver(Signals,Threshold))
      enterShort();
  }
}
```

Proper training requires some additional lines at the begin of the script:

```
set(PARAMETERS+LOGFILE+TESTNOW+PLOTNOW);
BarPeriod = 240;
LookBack = 500;
```

PARAMETERS is a flag that can be activated with the **set** function, just as the **LOGFILE** flag used in the last chapter. Several flags can be set by simply adding them in the **set** call. **PARAMETERS** instructs Zorro to adjust the parameters of the script in a training run and to use these adjusted values during the test run or trades. **TESTNOW** runs a test after training without the need to click on the **[Test]** button, and **PLOTNOW** generates a chart without the need to click on the **[Result]** button.

LookBack is the 'initialization time' that the strategy needs to calculate the initial values of indicators or functions. Only after that time period it can start trading. Zorro normally calculates the lookback time, so the user needs not to set it explicitly. But here the time depends on parameters changed by training. We must therefore specify it manually when we train a strategy. In this case, it's the 500 bars that the **FisherN** function requires. All other functions

require less bars, so we are on the safe side with 500. A too high lookback time is not harmful, except that it reduces the test period; a too low lookback time results in an error message.

```
if(Train) Detrend = TRADES;
```

The **if(Train)** condition only executes the following command in training mode. In this case, the default variable **Detrend** is set to the value **TRADES**. With **Detrend** you can manipulate change the price curve. For example, the curve can be tilted for removing a rising or falling trend. Or the trends can be reversed, or the curve can be randomly shuffled so that all the inefficiencies disappear from it. In this case, the **TRADES** mode is used, which causes long and short trades to achieve the same profit on average, regardless of the trend of the curve. Otherwise, if the curve had a long-term upwards trend, long trades would produce higher average profit than short trades, which could distort the training result. We only want to take this into account when training, the test should be performed done with the original price curve - hence the **if(Train)** condition.

The core of the training is calculating the trade signals with adapted parameters:

```
var Threshold = optimize(1,0.8,1.2,0.1);
LifeTime = optimize(100,50,150,10);
Trail = Stop = optimize(10,4,20,2) * ATR(100);
...
var RegimeThreshold = optimize(1.5,1.3,1.7,0.1);
```

Four parameters of the strategy are trained. The signal threshold is now determined by the **optimize** function and assigned to the variable **Threshold**. We can see that **optimize** is called with 4 parameters. The first (**1**) is the default value – that's normally the value used in the unoptimized version of the strategy. The next two numbers, **0.8** and **1.2**, are the upper and lower limits of the parameter. The fourth number, which can also be omitted, is the step width of the parameter. **Threshold** now runs through a range from **0.8** to **1.2** in increments of **0.1**. During the training process, Zorro will try to find the most robust value within that range. If the step width is omitted or zero, Zorro simply increases the parameter value by 10% for each optimization step.

The next training parameters are the life time of the trade, the factor for the stop-loss distance, and finally the market regime filter. By the way, the order of optimize calls in the script matters. Parameters that determine the trade

entry - in this case the bandpass cycle and the threshold value - should be trained first, then the exit parameters, as here the stop distance, and finally filters that prevent trading in certain market situations. Theoretically one could train much more parameters, for example the time period for the bandpass filter or the ATR function. But the more parameters we train, the higher the risk of overfitting the strategy to the historical price curve. This leads to strategies that pass a simple backtest with flying colors, but fail miserably in real trading as well as in more sophistcated backtests. We shall soon look into such test methods more closely. Until then, keep in mind that only a few essential parameters should be trained, and only within a reasonable parameter range.

To train the strategy, click **[Train]** and watch what the **optimize** calls do. After the training phase, which takes about one minute depending on the PC speed, a web page with diagrams like the following will pop up in your browser:

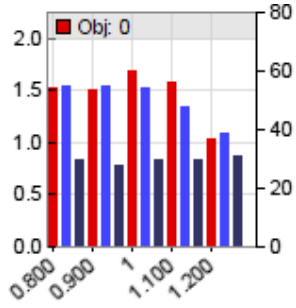


Fig. 27 – Signal Threshold (parameter 1)

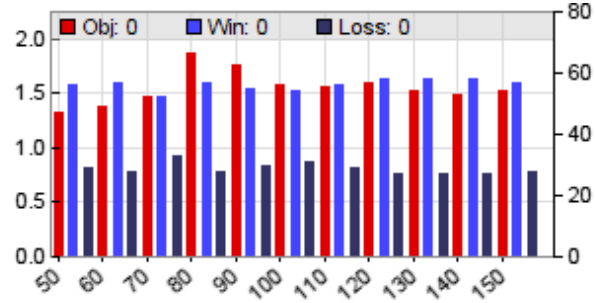


Fig. 28 – Trade maximum lifetime (parameter 2)

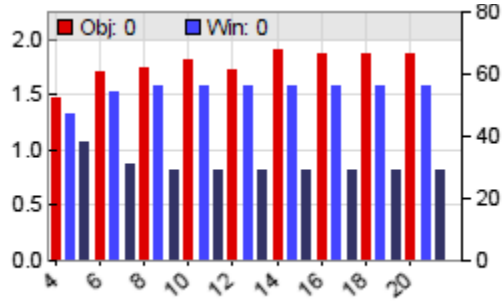


Fig. 29 – Stop factor (parameter 3)

These diagrams are produced by the training process when the **LOGFILE** flag is set. They show how the parameter values affect the outcome of the strategy. The red bars are the objective of the training period, i.e. the values that are optimized by the system. By default it's the **P**essimistic **R**eturn **R**atio (**PRR**) - the total profit divided by the total loss, multiplied by a 'penalty factor' that punishes parameter values with less trades. At a PRR above 1.0, the strategy is profitable even with the penalty factor.

The light blue bars are the number of profitable trades, the dark bars the number of lost trades. The horizontal axis is the parameter value at which the result was obtained. We can see that the strategy remains profitable – i.e. PRR is above 1 - over the entire range of the parameters. The threshold has the best performance at about 1.0, and the stop factor, the third parameter, berforms better with higher distance. Since almost all red bars end above 1.0 in the profitable area, the strategy seems to be robust and not very sensitive to parameter changes - at least with the price curve tested. The determined parameter values with the greatest robustness are stored in the **Data** directory in the file named **Alice2c_EURUSD.par** (the file name would be different for other assets) for later use when testing and trading.

Clicking **[Test]** shows that the training has indeed improved the strategy. But is this due to an adaption to the historical price curve, or to better parameters which would also produce more profit in live trading? This cannot be determined with a simple backtest. To trust this result without further examination would be a serious mistake.

Walk-forward optimization

Alice has trained the strategy with the price data from the last 6 years and then tested it with the same price data. This always leads to optimistic results - as if someone were preparing for an exam by already knowing the questions and memorizing the answers. There is another problem. In six years, markets are changing and trading strategies need to adapt. It is not recommended to keep a strategy unchanged for so many years. The strategy parameters should be adjusted at regular intervals to the market situation. Zorro can do this automatically, but how can we simulate this regular re-training in the backtest?

Walk-**F**orward **O**ptimization (**WFO**) puts the training process into the strategy itself. This allows - with certain limitations - to determine whether a strategy has the chance to achieve its backtest results even in live trading. At the same time, WFO can regularly adapt of the strategy parameters to the market situation. And it ensures that the price data used for testing is always 'out-of-sample', meaning it was not used for training.

To enable WFO, two lines are needed (**Alice2d**):

```
StartDate = 2010;
NumWFOCycles = 10;
```

With this setup the parameters are trained 10 times during the simulation. For this purpose, a 'time window' is shifted in 10 cycles over the simulation period. The window selects a section of the entire period. The section is divided into the lookback period, the training period and the subsequent test, as in the following figure:

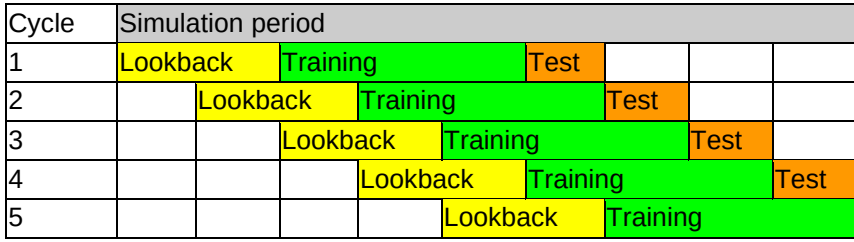


Fig. 30 - Walk-forward optimization, 5 cycles

The lookback time at the beginning of each window is used to gather the initial data for the indicators and functions. The training period produces the

parameters, which are then tested in the subsequent test period. This division ensures that each test always uses a fresh part of the price curve that has not already been used for the training of the parameters. The window is then moved ahead in time by the amount of the test period, and a new training / test cycle is started with a new section of the price curve. The last training cycle generates the parameters for the live trading and therefore has no subsequent test period.

The test period is now much smaller than the entire simulation period. Therefore, Alice has set the start date (**StartDate**) back to 2010 for getting enough time and price data. Since each test is preceded by a long training period, the first test cycle is now in 2014. If **StartDate** is not set, the simulation always starts 6 years before the current year.

After clicking on **[Train]**, the training process now takes a few minutes, since the simulation time is now more than 10 years and a complete optimization is performed for each of the WFO cycles. The trained parameters are stored in separate parameter files for each cycle. This way the test loads different parameters for every WFO section of the simulation. Due to the flags that Alice has set, test and chart plotting are automatically performed after the training:

Fig. 31 - WFO result

WFO can produce a worse or a better test result than a mere optimization with the whole price data. Using previously unseen, out-of-sample data reduces the test performance; adapting parameters to different market regimes

in any WFO cycle improves it. Normally, the worsening effects win. Here we get a similar or even slightly better result with WFO. The disadvantage, of course, is that the training process takes longer since it runs over many cycles, and that less price data is available for training. The 10 cycles chosen by Alice are a compromise: more cycles would shorten the training period per cycle, reduce the data available and make the training less effective. Less cycles bear the danger that the training periods extend over periods of different market regimes, which can make the trained parameters obsolete.

Self-training

Walk-forward optimization integrated into the script can generate a basically 'parameter-free' strategy - a strategy that does not depend on a once-optimized parameter set, but generates its own parameter values by regularly training itself. The WFO process does not only test the strategy, but also the quality of the parameter adaption.

For regularly adapting its parameters to the market in live trading, Alice has added the following lines to the script:

```
ReTrainDays = 147;
if(ReTrain) {
  UpdateDays = -1;
  SelectWFO = -1;
}
```

ReTrain is true when a re-training run is started during live trading, either automatically every 147 days, or manually by clicking the **[Train]** button of a trading Zorro. The number of days (**RetrainDays**) between retraining runs should be identical to the WFO test period. It is displayed in the performance report. **UpdateDays** is a time period after which new price data are automatically downloaded from the broker to update the price curve for training. If **UpdateDays** is set to **-1**, price data are always downloaded. **SelectWFO** causes Zorro not to re-train the entire simulation period, but only the given WFO cycle. In this case, it is the last cycle (-1) that covers the new price data and generates the parameter set for live trading.

The retraining mechanism updates the parameters regularly and this way makes the strategy independent of their initial values. Automated retraining requires Zorro S, but you can also retrain with the free Zorro version by regularly downloading the newest price data and manually running a training

session with another Zorro instance. The trading Zorro will then automatically switch over to the re-trained parameters at the next bar period.

Beware of bias

So you have now developed a strategy and observe huge profits in a 5-years WFO test. Open a broker account, fire up the software, click **[Trade]**, wait 5 years, and you'll be rich. Or not?

The test applies the strategy to a historical price curve. This price curve contains the inefficiencies that you exploit in your strategy for getting nice profits. But how do you know that profits are based on real market inefficiencies that are likely to re-appear in future price curves, and not on random patterns in historical data that will not repeat?

The likelihood of the strategy functioning with a real market depends on how it was developed and optimized. As we have seen, backtest results are almost worthless when they are based on 'in-sample' data that was also used for the training. This always results in optimistic test results. Sure, WFO allows to use different price curves for training and for testing. However, you will usually choose your strategy just because of their positive WFO test. This means that the 'out-of-sample' price data in the test period are not so out of sample anymore, since they were used for selecting the best strategy. The same problem arises when you examine multiple assets and choose the most profitable of them for a particular strategy. All result-based selection processes distort the result. Therefore the backtest result is never completely reliable. Every backtest lies - the question is only to what extent it does so.

The problem worsens when the strategy performance is very sensitive to small differences in price data and parameters. This is especially bad if the strategy behavior depends on previous results, such as a martingale system that increases the investment at any loss. Or if it depends heavily on the start and end time of the bars, as is the case with strategies that compare prices to specific market times. Or if it reinvests profits, for instance by trading with 1% of the accumulated capital. Such systems can have a chaotic profit curve and a strongly fluctuating end result, which greatly increases the likelihood that the system owes its positive backtest to chance.

Since there is no mathematical formula to calculate the outcome of a strategy, backtesting is the only option. For being able to halfway rely on the test results, one must know the effects that cause biased results. Here is a list of the most important of these biasing effects. Some can be completely avoided by

appropriate test procedures, others can be estimated at least and can be deducted from the result:

Curve fitting bias. Each strategy is adapted in some way to the historical price curve, but the question is whether all its profitability is due to this adaption. This would cause largely too optimistic results and the illusion that a strategy is profitable when it isn't. By appropriately selecting the optimization method and the test and training periods, curve fitting bias can be minimized.

Market fitting bias. Caused by a poor choice of the "out-of-sample" data, for example by the usual statistical method of selecting random samples from the price data. The test then uses data that is, although out-of-sample, possibly from the same market situation as in training, and thus produces too optimistic results. Market fitting bias can be avoided by selecting test data always from *after* and never from *before* the training data.

Peeking bias. The trade algorithm knows in the backtest some aspect of the future. This is usually caused by improper trading software, especially when the profit was determined by trade simulation in R, Excel or with self-written programs. It is very easy to make small programming errors that cause peeking bias. A different sort of peeking bias is caused by calculating trade volumes with capital allocation factors that are generated from the whole test data set. To prevent this, backtests should be generally run at first with fixed volumes or lot sizes.

Data mining bias. Caused not necessarily by data mining, but already by the mere act of developing a strategy with historical price data. Selecting the most profitable from a pool of possible algorithms, or selecting the most profitable asset from several asset combinations, can already cause large bias in the test result. This bias is often confused with curve fitting bias, but also affects strategies that have no parameters at all to be fit. The only way to completely avoid data mining bias is not testing strategies. Since this is not practical, there are methods to at least estimate the effect of data mining bias on the result - for instance comparing the performance with a distribution of strategy performances with randomized (shuffled) price data.

Trend bias. Affects 'asymmetric' strategies that use different algorithms, parameters, or capital allocation factors for long and short trades. An extreme case are strategies that enter only long or only short trades. Price trends in the test or training period can then greatly distort the result. This can be prevented by detrending the trade signals or the trade results, as in Alice's script by setting the **Detrend** variable. Another sort of trend bias can be caused by rollover profit or loss in combination with long-term trades.

Granularity bias. A consequence of the different price data resolution in test and in real trading. Historical prices only change once per bar, while real time price quotes can arrive many times per seconds. This causes stops and profit targets to be triggered at different moments. For reducing granularity bias, let the system also evaluate the proce curve inside a bar, for instance by setting Zorro's **TICKS** flag. The disadvantage is, of course, that the test can last a lot longer.

Sample size bias. Causes a dependency of the test result on the length of the test period. The magnitude of fluctuations in the equity curve, and therefore the drawdown of a system, is usually proportional to the square root of the test time. This produces pessimistic results for long test periods, too optimistic results for short periods. Therefore, test results should be projected onto a fixed time to be comparable. Often a period of three years is used.

All these problems sound as if a backtest is more or less like reading coffee. It is not so bad, however, because there are two good news. Firstly, when randomly setting strategy parameters, the parameter set that achieves the better backtest result is also likely to achieve the better live trading result (which does not mean that it will be profitable at all!). Secondly, all the mentioned bias effects diminish with increasing test data. The more trades a backtest executes, the more the result will be bias free and resemble live trading. Therefore it makes sense to get as many trades as possible out of the available data. Despite all the biases, backtest results are still useful when some rules are observed.

Rules of thumb

Zorro has built-in functions to estimate, minimize, or eliminate many test problems – so there's no reason why not to use them, even if it requires a line or two in the script. Many cliffs can be circumnavigated by a rational approach to strategy development. Here are some practical tips on training and optimization.

Before a strategy is trained for the first time, run a test with the default parameters. Check the log file for error messages such as skipped trades or incorrect numbers. In this way you can detect and correct errors, because the training itself will not generate any error messages. In training, trades are simply not executed if something is wrong.

When developing the strategy, consider which parameters are the best candidates for optimization. Train those with a significant impact on the outcome of the strategy. Look at the parameter diagrams: They provide valuable information about the effect of the parameters. Only use strategies whose parameters in the diagrams produce "wide hills", no narrow peaks, single lines, or a chaotic diagram.

Keep the trade algorithm as simple as possible and do not use too many parameters, filters or additional conditions. A strategy should preferably have no more than 3 relevant parameters for opening trades and maybe 3 more for closing. Too many parameters increase the likelihood of curve fitting bias and lead to poor results in the WFO process and in live trading.

First train the entry parameters (which determine the opening of trades), then the exit parameters (which are responsible for closing). If your exit rules are complex, optimize the entry parameters with a simplified exit procedure, such as a simple stop loss. Otherwise the result is distorted by the exits' influence on the profit. In the script, you can use the variable **ParCycle** to identify the the parameter that is currently being optimized, and depending on that temporarily suspend the exit rules.

Choose large steps for the optimize function. This accelerates the training and reduces the risk of producing local maxima and generating over-adapted strategies.

Use the same parameters for short and long trades, unless the asset behaves very asymmetrically. Asymmetric assets show a different behavior with rising and falling prices. This is the case for equities and equity indices, but not for forex. Different parameters for short and long trades can be used here. Use different Algo identifiers with ": S" (short) or ": L" (long) at the end.

Do not use too short and not too long WFO cycles. Too short cycles contain too few trades, so that parameters can not be trained effectively and the result is randomly dependent. Too long cycles will worsen the result if the inefficiency changes or disappears within the cycle and the parameters do not adapt fast enough to the changed market.

Ensure enough trades. Systems that only trade three times a year can not be backtested. The rule of thumb is: at least 30 trades per WFO cycle and per parameter. This means that for 3 parameters, you need at least 90 trades per cycle. More are, of course, better. For critical cases, Zorro offers a trick: By oversampling the price curve (variable **NumSampleCycles**) you can create different variants of the curve, in which most inefficiencies are retained. This

increases the number of trades and thus the quality of training and testing without extending the training period.

7 tricks for counter trend strategies

► Detect and exploit cycles in price curves. For this, make yourself familiar with spectral analysis and transformation functions. A **bandpass filter** emphasizes cyclic effects and removes trend and noise from a price curve. The **Fisher Transform** compresses a curve to a normal distributed range.

► Look into all relevant signals. Use **plot** functions for displaying the curves of signals and indicators.

► **Train** parameters for making a strategy more robust and for checking the effect of parameters on the result.

► It makes sometimes sense for training and even testing to **eliminate long-term trends** from the price curve or from test results.

► **Walk Forward Optimization** simulates trading under realistic conditions with out-of-sample price data. Use it whenever you train parameters.

► Trained strategies should be **re-trained** in regular intervals for adapting them to new market conditions.

► Test results are affected by many sorts of **biases**. Even WFO is no guarantee to replicate the test results in live trading. Make yourself familiar with all potential traps of testing.

5 The money breeder

Bob: "I got it! This is it!"
Alice: "Bob, why are you running naked through the streets?"
Bob: "Huh? Oh, I was just sitting in my bathtub when I suddenly got this brilliant idea. I need you to program it immediately. This will solve my trading problems!"
Alice: "Problems? I thought you were already getting rich by automated trading?"
Bob: "It started well, but then it went all wrong. You know, from all the systems you programmed for me, the counter trend system worked best."
Alice: "I figured that."
Bob: "And from all assets, I found it had the most profit with the EUR/USD. So I invested all my money in that system trading with EUR/USD."
Alice: "Oh no!"
Bob: "Yes! And it worked like a charm! My account went up from $100,000 to $250,000 in six months. I naturally began to think about what do do with all that money. I had already ordered a Porsche and a golden bathtub."
Alice: "Naturally."
Bob: "But then all of the sudden, your system started losing money. My account went down and down and didn't stop going down."
Alice: "How much?"
Bob: "It lost $100,000 in a single month. I'm now back at $150,000, and it's staying there since weeks."
Alice: "Wins and losses follow a random sequence. Any system will eventually meet a loss streak of any length and depth - that's mathematically certain."
Bob: "I know, I know, it's a drawdown and I must just sit it out. But I don't have the nerve for that. Every morning I'm looking at the PC screen and my account is either going down, or not going up!"
Alice: "Put a blanket over your PC."
Bob: "I got a better idea. That's why I was on the way to you. What if the system traded with many assets at the same time?"
Alice: "Indeed, that could reduce the variance when the assets uncorrelated."
Bob: "I have no idea what you mean with that. But when EUR/USD is in a drawdown, chances are that another pair, like USD/JPY, is still profitable. So I lose with one, but win with the other. And I can do this with a portfolio of many assets."

Alice: "Certainly."
Bob: "And can't you also combine your systems into one? I mean a super system that goes with the trend and against the trend at the same time? So I win when assets are trending and I also win when assets are cycling? Meaning I win all the time?"
Alice: "You probably won't win all the time when all the different price curves are still somewhat correlated. But it can reduce your drawdowns."
Bob: "Well, then program it. I'll sell my Porsche and pay you well."
Alice: "Good. I'll start at once. Now better go home and put some clothes on."

A forex portfolio system

Alice got the task to write a script that trades simultaneously with multiple assets and strategies. This is her first version (script **Alice3a**, agreed fee: $5000):

```
function tradeCounterTrend()
{
   TimeFrame = frameSync(4);
   vars Prices = series(price());
   vars Cycles = series(BandPass(Prices,30,2));
   vars Signals = series(FisherN(Cycles,500));
   var Threshold = optimize(1,0.8,1.2,0.1);

   LifeTime = 4*optimize(100,50,150,10);
   Trail = Stop = optimize(10,4,20,2)*ATR(100);
   MaxLong = MaxShort = -1;

   var Regime = FractalDimension(Prices,100);
   var RegimeThreshold = optimize(1.5,1.3,1.7,0.1);
   if(Regime > RegimeThreshold)
   {
    if(crossUnder(Signals,-Threshold))
      enterLong();
    else if(crossOver(Signals,Threshold))
      enterShort();
   }
}

function tradeTrend()
```

```
{
   TimeFrame = 1;
   vars Prices = series(price());
   vars Trends = series(Laguerre(Prices,
    optimize(0.05,0.02,0.15,0.01)));

   Stop = optimize(10,4,20,2)*ATR(100);
   Trail = 0;
   LifeTime = 0;
   MaxLong = MaxShort = -1;

   var MMI_Period = optimize(300,100,400,100);
   vars MMI_Raws = series(MMI(Prices,MMI_Period));
   vars MMI_Avgs = series(SMA(MMI_Raws,MMI_Period));

   if(falling(MMI_Avgs)) {
    if(valley(Trends))
      enterLong();
    else if(peak(Trends))
      enterShort();
   }
}

function run()
{
   set(PARAMETERS+LOGFILE+TESTNOW+PLOTNOW);
   StartDate = 2010;
   EndDate = 2018;
   BarPeriod = 60;
   LookBack = 4*500;
   if(Train) Detrend = TRADES;

   NumWFOCycles = 10;
   NumCores = -1;
   ReTrainDays = 147;
   if(ReTrain) {
    UpdateDays = -1;
    SelectWFO = -1;
   }

   while(asset(loop("EUR/USD","GBP/USD")))
   while(algo(loop("TRND","CNTR")))
   {
    if(Algo == "TRND")
      tradeTrend();
```

```
    else if(Algo == "CNTR")
      tradeCounterTrend();
   }
}
```

The strategy is now split into 3 different functions: **tradeTrend** for trend trading, **tradeCounterTrend** for counter trend trading, and the **run** function that sets up the parameters, selects the assets, and calls the two trade functions.

The **tradeCounterTrend** function is almost the same as in the last workshop. Only at the begin of the function Alice has added the line

```
TimeFrame = frameSync(4);
```

What does this mean? The trend trading strategy from workshop 4 used 60-minutes bars, while counter trend trading was based on 240-minutes bars. When we trade both together in the same script, we'll need both types of bars. The **BarPeriod** variable can not be changed at runtime, since it determines the sampling of the price curves. But the **TimeFrame** variable does the job. Alice has set **BarPeriod** to 60 minutes, so it needs a **TimeFrame** of 4 bars for getting the 240 minutes period for the counter trend strategy. **TimeFrame** affects all subsequent trade, price, and series function calls, and all indicators using those series, such as the **ATR** function.

Simply setting **TimeFrame = 4** would normally also do the job, but the **frameSync** function mimicks more accurately the behavior of a 240 minutes bar period. Bar periods are synchronized to the time of day, so a 4-hour bar period always ends at 0 am, 4 am, 8 am, 12 pm, 4 pm, or 8 pm (plus an optional **BarOffset**). But due to weekends and gaps in the historical data, four 1-hour bars are not always equal to four hours. The **frameSync** function checks the current time and makes sure that the time frames end at the same time as 4-hour bar periods. Even then, **TimeFrame** is not always 100% identical to a longer bar period. There are a few subtle differences that are listed in the Zorro manual. We need not care about them here, with one exception: Since the **LifeTime** variable is in **BarPeriod** units, its value must now also be multiplied with 4 for getting the same trade duration as in Alice's previous strategy.

The **tradeTrend** function uses the same algorithm as in Alice's first strategy. The **TimeFrame** variable is set to 1, so this strategy is still based on a 60-minutes bar period. The **Laguerre** parameter, the stop loss factor and the

MMI length are now optimized with the method explained previously. Alice has also explicitly set the **Trail** and **LifeTime** variables to 0:

```
Trail = 0;
LifeTime = 0;
```

Trend trading works best when profitable trades last a long time; trailing would stop them too early. However, the **Trail** variable was already set in the **tradeCounterTrend** function. If Alice had not reset **Trail** and **LifeTime** to 0 (meaning no trailing and no time limit), they would keep their last values, **tradeTrend** would trail too, and its profits would probably (try this!) go down. When predefined variables are used anywhere in the script, make sure that they have the right value at any place where they are needed. This is a typical source of mistakes, so make it a habit to always look inot the trade log to be sure that trades behave as they should.

The first lines of the **run** function are similar to the previous script, only **BarPeriod** is now at 60 and consequently **LookBack** at 2000 for getting the same 500 4-hour periods. Only one further line has been added:

```
NumCores = -1;
```

Modern processors have several CPU cores, but normally only one per running program is used. The **NumCores = -1** statement tells Zorro to use for training all available cores except one. Each WFO cycle is assigned to one core. The one remaining core can then perform other tasks, such as checking for email. The training process is several times faster this way - and that makes sense, because the training of a portfolio system with many components can take a long time. This feature is unfortunately only available in Zorro S, not in the free Zorro version; the variable **NumCores** is then simply ignored.

The main strategy looks a bit strange:

```
while(asset(loop("EUR/USD","GBP/USD")))
while(algo(loop("TRND","CNTR")))
{
  if(Algo == "TRND")
    tradeTrend();
  else if(Algo == "CNTR")
    tradeCounterTrend();
}
```

We have two 'nested' **while** loops, each with two nested function calls. Let's untangle them from inside out:

```
loop("EUR/USD","GBP/USD")
```

The **loop** function takes a variable number of parameters, and returns one of them every time it is called. On the first call it thus returns the first parameter, which is the string **"EUR/USD"**. We have learned that a string is a variable that contains text instead of numbers. Strings can be passed to functions and returned from functions just as numerical variables. If the **loop** function in the above line is called the next time, it returns **"GBP/USD"**. And on all further calls it returns **0**, as there are no further parameters.

```
asset(loop("EUR/USD","GBP/USD"))
```

The string returned by the **loop** function is now used as parameter to the **asset** function. This function selects the traded asset, just as if it had been choosen with the **[Asset]** scrollbox. The string passed to **asset** must correspond to an existing asset with available price history. If **asset** is called with 0 instead of a string, it does nothing and returns **0**. Otherwise it returns a nonzero value. This is used in the outer **while** loop:

```
while(asset(loop("EUR/USD","GBP/USD")))
```

This loop is repeated as long as the **while** expression, which is the return value of the **asset** function, is nonzero (a nonzero comparison result is equivalent to 'true'). And **asset** returns nonzero (= true) when the loop function returns nonzero. Thus, the **while** loop is repeated exactly two times, once with **"EUR/USD"** and once with **"GBP/USD"** as the selected asset. If Alice had wanted to trade the same strategy with more assets - for instance, also with commodities or stock indices - she only needed to add more arguments to the loop function, like this:

```
while(asset(loop("EUR/USD","USD/CHF","GBP/USD","AUD/USD","XAU/USD",
"USOil","SPX500","NAS100",...)))
```

and the rest of the code would remain the same. However, Alice does not only want to trade with several assets, she also wants to trade different algorithms. For this, nested inside the first while loop is another while loop:

```
while(algo(loop("TRND","CNTR")))
```

The **algo** function basically copies the string parameter into the predefined **Algo** string variable that is used for identifying a trade algorithm in the performance statistics. It also returns 0 (= false) when it gets 0, so it can be used to control the second **while** loop, just as the **asset** function in the first **while** loop. Thus, for every repetition of the first loop, the second loop repeats 2

times, first with **"TRND"** and then with **"CNTR"**. As these strings are now stored in the **Algo** variable, Alice uses them for calling the trade function in the inner code of the double loop:

```
if(Algo == "TRND")
  tradeTrend();
else if(Algo == "CNTR"))
  tradeCounterTrend();
```

The **if** condition checks if the **Algo** string is identical to a directly given string constant (**== "TRND"**) and returns nonzero, i.e. true, in that case. So when **Algo** was set to **"TRND"** by the **algo** function, **tradeTrend** is called; otherwise **tradeCounterTrend** is called. Due to the two nested loops, this inner code is now run four times per **run** call:

- First with **"EUR/USD"** and **"TRND"**
- Then with **"EUR/USD"** and **"CNTR"**
- Then with **"GBP/USD"** and **"TRND"**
- And finally with **"GBP/USD"** and **"CNTR"**

Those are the 4 possible asset/algorithm combinations - the **components** - of the strategy. If Alice had instead entered 10 assets and 5 algorithms in the loops, we had 10*5 = 50 components and the inner code would be run fifty times every bar. Keep in mind that all parts of the script that are affected by the asset - for instance, retrieving prices or calling indicators - must be inside the **asset** loop. This is fulfilled here because anything asset related happens inside the two called trading functions.

Now it's time to train the system.

Portfolio analysis

Because we have now 4 components and 4x more bars, the training process (click **[Train]**) will take much longer than in the last chapter, at least with the free Zorro version with no multicore support. As soon as it's finished, let's examine the resulting parameters. Use the script editor to open the **Alice3a_1.par** file in the **Data** folder. This file contains the optimized parameters from the first WFO cycle. It could look like this:

```
EUR/USD:TRND 0.039 12.05 287 => 2.536
EUR/USD:CNTR 1.000 100.0 7.93 1.498 => 0.825
GBP/USD:TRND 0.110 12.00 218 => 0.797
GBP/USD:CNTR 1.012 79.9 16.02 1.520 => 1.096
```

As we can see, parameters for every asset/algo combination are optimized separately. Each line begins with the identifier for the component, consisting of asset and algorithm separated by a colon. Then follows the list of parameters. This allows Zorro to load the correct parameters at every **asset()** and **algo()** call. We can see that the **tradeTrend()** function uses three parameters and the **tradeCounterTrend()** function four, and their most robust values are different for the **"EUR/USD"** and the **"GBP/USD"** assets. By the way, if a **'+'** sign appears in front of a parameter, its most robust value has been found at the very start or end of its range. This hints that the range might be too small and should be extended. The last number behind the **"=>"** is not a parameter, but the result of the **objective** function of the optimization run; it is for information only and not used in the strategy.

After the training, a test run is automatically executed, and the result plotted:

Fig. 32 – Portfolio result

You can see in the chart above that now two different algorithms trade simulaneously. The long green lines are from trend trading where positions are hold a relatively long time, the shorter lines are from counter-trend trading. The combined strategy does both at the same time. It generated about 130% annual profit with $200 minimum capital requirement in the walk forward test. This equity curve contains some of the typical characteristics that you'll encounter with most profitable automated systems:

Boredom: There are long time periods with little trading and not much equity changes, as in 2015.

Profit bursts: In spring 2016 a sequence of lucky trades produced large gains within a relatively short period.

Drawdowns: In winter 2015 and summer 2016 a trader needed strong nervers to stomach the periods of steady losses.

Now look at the end of the performance report that pops up when clicking **[Result]** (it is also stored under **Alice3a.txt** in the Log folder). You can see how the separate asset/algo combinations - the components of the strategy - performed during the test:

```
Portfolio analysis  OptF  ProF  Win/Loss  Wgt%  Cycles

EUR/USD avg         .113  2.57   64/49    80.0  XXXXX/XXX
GBP/USD avg         .058  1.21   89/120   20.0  XXXXXXX/X

CNTR avg            .141  2.19   65/35    52.0  XXXXX/XXX
TRND avg            .087  1.46   88/134   48.0  XXXXXXXXX

EUR/USD:CNTR        .209  2.95   44/19    44.9  /X/\//XX/
EUR/USD:CNTR:L      .185  2.62   23/7     19.2  /\/\/////
EUR/USD:CNTR:S      .243  3.30   21/12    25.8  ///\//\/
EUR/USD:TRND        .125  2.25   20/30    35.0  \/X/\/X\X
EUR/USD:TRND:L      .071  1.55    9/15     8.4  \/\/\//\\
EUR/USD:TRND:S      .180  3.09   11/15    26.7  \.//\/\\X
GBP/USD:CNTR        .073  1.34   21/16     7.0  X/XXX/X/X
GBP/USD:CNTR:L      .117  1.68   10/7      6.2  //\\////\
GBP/USD:CNTR:S      .018  1.07   11/9      0.8  \.//\/\./
GBP/USD:TRND        .040  1.17   68/104   13.0  X\\XXXX/X
GBP/USD:TRND:L      .000  0.95   30/57    -2.0  /\\\\\//X
GBP/USD:TRND:S      .097  1.46   38/47    15.0  \\\//\//
```

The result is broken down separately for each asset, algorithm, and short (**":S"**) or long (**":L"**) trades of the component. The numerical value in the first column (**OptF**) is a factor for reinvesting profits, to which we will return soon. **ProF** is the profit factor, **Win/Loss** is the number of won and lost trades, and **Wgt%** is the contribution of the component to the total profit of the portfolio in percent. The last column, **Cycles**, indicates how well the component has fared in the individual WFO cycles. A rising bar **/** stands for gain, a falling bar **** for loss, an **x** for both.

We see that the highest contribution to the result – more than 40% - was generated by the **EUR/USD** system. Trend trading with **GBP/USD** has been the worst performer and produced even a net loss for long trades. This raises obvious questions: Would it not make sense to invest more money into the profitable components and less money into the less profitable? And would it not increase our profit when we reinvest the gains and in this way increase the trade volume gradually? We see in the report that in five years the system had generated approximately $1200 in profits from the $200 minimum initial capital. How much more could we achieve with the same strategy and the same capital when we wisely invest our winnings?

Money Management

Money management is an algorithmic method for distributing your profit to a portfolio of assets and trading algorithms in such a way that a) the return is maximized and b) the risk is minimized. The obvious problem: a) and b) are mutually exclusive. So a compromise is asked for. Consequently, many different money management methods and philosophies exist. The simplest method taught in trading books is: "Invest 1% of your balance per trade". This is, in fact, not a good idea.

Firstly, it is unclear what "invest" does actually mean for a financial derivative. The investment would normally be the price of the purchased asset. But this approach does not make much sense with high leverage accounts, especially in the case of forex trading. You would need a capital of one million dollars to buy only 1 standard lot of 10,000 currency units according to the 1% rule. Many books recommend using the "risk" of a trade as a measure for investment. The maximum loss – basically the distance of the stop loss from the current price – is assumed as the risk. Apart from the fact that the real loss can be much higher than the stop distance - as EUR/CHF traders had to learn on 15 January 2015 – would this make the investment dependent on the stop-loss position. Which would result in nonsensical trade volumes, especially in strategies that use a very distant or no stop loss at all. Sometimes it is even recommended to do it the other way around and set the stop loss depending on your capital. This advice would in fact lead to its quick disappearance, and this way indeed solve the issue once and for all.

The most useful measure for the investment is the margin, the amount that the broker will keep as a deposit for each open trade. Without leverage, this would be exactly the price of the asset. The margin has not much to do with the risk of a trade, but it at least ensures that the amount of the investment always corresponds to the purchased volume. This is particularly important

for strategies that hedge the risk by opening a position with a correlated asset in the opposite direction.

The second problem: 1% of whatever are usually either too little for a decent return, or too much for an acceptable risk. And sometimes even both at the same time. As a matter of fact, traders who always invest 1% will inevitably one day face an empty account - even with a profitable strategy! As we shall see below, this is statistically unavoidable - the question is only how quickly this happens. But if 1% of your balance is too little and too much at the same time, what the heck are you going to invest?

Financial analyst Ralph Vince has developed a computer algorithm that calculates the optimal percentage of reinvesting the profit from the balance curve of each component. This percentage is referred to as "Optimal-F" (**OptF** column in the table above). Multiply Optimal-F with the capital generated so far, and you get the maximum amount to invest in a trade. Normally you try to stay significantly below that amount, for the usual reason: Optimal-F was derived from the historical balance curve, and there is no guarantee that this curve will continue in the future in the same way. Exceeding the maximum investment has much worse consequences than staying below it. In order to remain on the safe side, hedge funds managers often reinvest about 50% of the Optimal-F value.

The next question is which part of the profit is at all available for reinvesting. The capital requirement for a strategy depends on the expected maximum drawdown. However, this is a parameter that increases over time. In a 15-years backtest, a system always has a larger maximum drawdown than the same system in a 5-years backtest. The longer you trade, the higher the probability of a long loss series and the worse the drawdown to be expected. You must therefore add an ever-increasing amount to the account in order to compensate the maximum drawdown and keep the same distance from the margin call. You can not reinvest this amount in trades.

The drawdown to be expected can be estimated with a diffusion model. It's growing approximately[1] with the square root of the trading time. Therefore, each trading system will suffer a drawdown of any depth if you wait long enough. The drawdown naturally also increases with the lot size: with double trade volume you also get the double drawdown. So if you always invest a fixed percentage of your balance, the drawdowns grow proportionally to the balance. At the same time they grow proportionally to the square root of time. As both effects sum up, the drawdowns grow faster than the balance[2] – and therefore will inevitably reach and pass it one day. Which means margin call. Therefore, the usual 1% recommendation is a sure way to blow your account sooner or later, even with the most profitable system. And 'sooner or later' is not always some day in the far future, but often more sooner than later.

The obvious solution is to not reinvest a fixed percentage of the balance, but an amount proportional to the square root of the trading time. Since the capital of a profitable strategy grows linearly with time, this is equivalent to the square root of capital growth. The rest is not reinvested, but remains on the account and compensates for the increase in the drawdowns. This is the **square-root rule**, which we will often encounter in the rest of this book[3]. So if your capital has doubled, increase the trade volume only by a factor of about 1.4 (the square root of 2), which is about 40%. For example, you traded at 50 EUR margin per position. Your account has doubled from the initial 1000 EUR to 2000 EUR. Now you can increase the margin per trade to 70 EUR (1.4 * 50 EUR) to reinvest the maximum profit.

An alternative method would be to determine the Optimal-F value not only when developing the strategy from historical data, but also to recalculate it

[1] This is in fact only correct for a profit-neutral system that wins and loses the same amounts. But those details won't matter much here, since the square root rule is a good approximation for drawdown growth of all systems.
[2] When investing linearly, drawdowns grow proportionally to the balance raised to the power of 1.5.
[3] There are trading systems where the square-root rule does not apply. If drawdowns are small compared with the invested margin, you can invest linearly. We will discuss two of those systems in further chapters.

permanently from the real balance curve since the beginning of trading with this system. The Optimal-F value will then gradually decrease as the maximum drawdown increases. Then the square root rule would be unnecessary, since for reinvesting the capital can be multiplied directly with Optimal-F. However, this is only practicable if the same strategy is traded unchanged for a very long time.

Enough of theory. Alice has now changed the **run** function for optimal capital distribution and reinvesting profits (script **Alice3b**):

```
function run()
{
   set(PARAMETERS+FACTORS+LOGFILE+TESTNOW+PLOTNOW);
   StartDate = 2010;
   EndDate = 2018;
   BarPeriod = 60;
   LookBack = 4*500;
   Capital = 10000;
   if(Train) Detrend = TRADES;

   NumWFOCycles = 10;
   NumCores = -1;
   ReTrainDays = 147;
   if(ReTrain) {
    UpdateDays = -1;
    SelectWFO = -1;
   }

   while(asset(loop("EUR/USD","GBP/USD")))
   while(algo(loop("TRND","CNTR")))
   {
    Margin = 0.01 * (Capital+ProfitClosed);
    if(Algo == "TRND")
      tradeTrend();
    else if(Algo == "CNTR")
      tradeCounterTrend();
   }
}
```

There are three changes. At the begin of the script, Alice has now also set the **FACTORS** flag:

```
set(PARAMETERS+FACTORS+...
```

One **set()** call can set several flags at the same time by just adding them with '+' (or '|'). **FACTORS** lets Zorro calculate OptimalF factors. **[Train]** generates now not only parameters, but also factors separately for every strategy component. The factors are generated in the last optimization run and stored in **Data\Alice3b.fac**. It's a simple text file again, so all factors can be examined and edited with a text editor.

Re-training the strategy only uses the last WFO cycle; this is not enough for generating new OptimalF factors. Therefore, Alice resets the FACTORS flag when the system is retrained:

```
if(ReTrain) {
   UpdateDays = -1;
   SelectWFO = -1;
   reset(FACTORS);
}
```

Alice has now also invested $10000 initial capital:

```
Capital = 10000;
```

At first Alice wants to try the usual 1% method:

```
Margin = 0.01 * (Capital+ProfitClosed);
```

Margin is a variable that determines the amount invested per trade. It's a percentage of the real trade volume - for instance 1% at 1:100 leverage - that the broker keeps as a deposit. If Margin is left at its default value, Zorro always buys 1 lot, the minimum allowed trade size. The higher the margin, the higher the number of lots and the higher the profit or loss.

ProfitClosed is the sum of the profits of all closed trades of the portfolio component. **Capital+ProfitClosed** would be our current account balance if only this component would have been traded. This amount is multiplied with 1% - thus, with the number **0.01** – and used for the margin of the next trade.

Now click **[Train]**. In addition to the parameters, OptimalF factors are now also generated and stored in the **Alice3b.fac** file in the **Data** folder. The content of this text file looks similar to the above section from the portfolio analysis. The **OptF** values in the first column are read when testing or trading, and are then available to the script as **OptimalF** variables. They usually determine the trade volume of the relevant portfolio component. If a factor

has the value 0, the component was not profitable in the training run, and should better not be traded.

This is the result after training and testing:

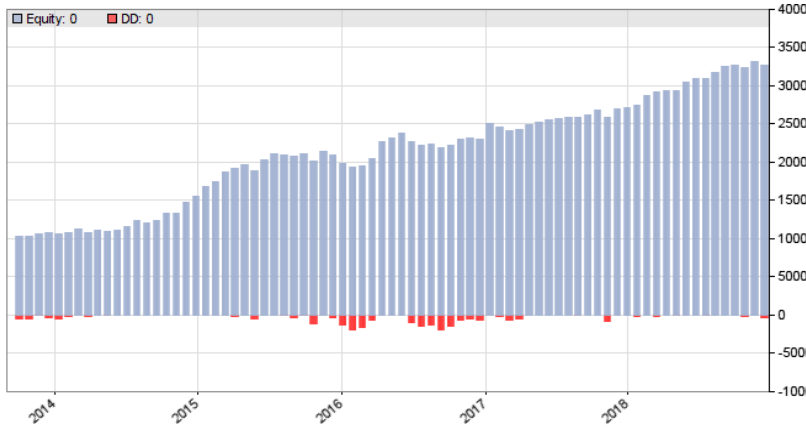


Fig. 33 – Reinvesting, 1% method

The system achieves a CAGR of approx. 25%, equivalent to a tripling of the initial capital at the end of the simulation. CAGR (**C**ompound **A**nnual **G**rowth **R**ate) is the average annual growth of the invested capital. Our initial $10.000 will grow with the 1% method on average to $12,500 after the first year, to $16,250 after the second and so on. Zorro displays the CAGR value instead of the Annual Return when the **Capital** variable is set to an initial capital. CAGR then refers to this capital, while the normally much higher annual return always refers to the calculated minimum capital.

The 25% growth, although achieved at the cost of a possible margin call sometime in the future, are certainly better than what Bob would get on a savings account. Still, they do not impress Alice much. She now tries the hedge fund method and reinvest the half of the OptimalF percentage:

```
Margin = 0.5 * OptimalF * (Capital+ProfitClosed);
```

OptimalF is the already mentioned optimal investment factor that Zorro has calculated from the balance curve of the backtest, using the Ralph Vince algorithm. This factor is adjusted the use of **Margin** as an investment measure. After changing the **Margin** line in the script, you only need to click **[Test]** since the strategy was already trained and modifying **Margin** has no effect on the trained parameters and **OptimalF** values. This is the result:

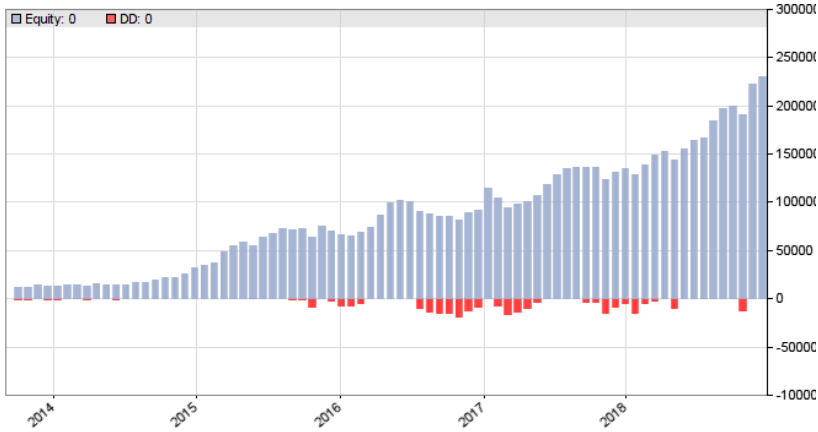


Fig. 34 – Reinvesting, hedge fund method

The CAGR has now risen to over 80%, and Alice's $10,000 have multiplied with twenty up to the end of the simulation. No wonder some of the funds have grown obscenely rich. However, this method of maximizing the profits produces long and dangerous drawdowns – often much worse than with this. Reinvesting in this way requires strong nerves. And maybe also a packed flight case in case some angry investors are after you since their money vaporized in a margin call...

Finally, Alice tests the square root method:

```
Margin = 0.5 * OptimalF * Capital * sqrt(1+ProfitClosed/Capital);
```

Again, half the **OptimalF** value is invested, but this time only the square root of the profit is available for reinvesting. For this purpose, the growth factor is calculated, which is the expression **(1+ProfitClosed/Capital)**. Assuming that the portfolio component has now generated $5,000 profit with the $10,000 initial capital. The growth factor then has the value **(1+10000/5000) = 1.5**, meaning that the initial capital has grown by a factor of 1.5. From this value, the square root is calculated (the square root of 1.5 is about 1.22) and then multiplied by half the investment amount (**0.5 * OptimalF * Capital**).

The result does not look bad:

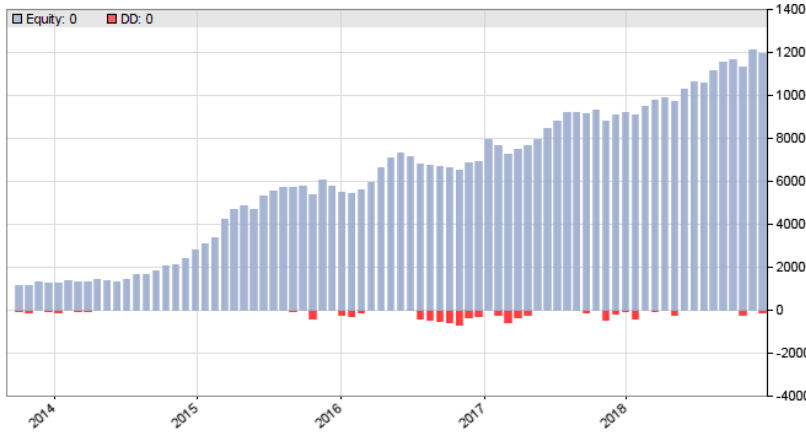


Fig. 35 – Reinvesting, square root method

The CAGR has declined somewhat, but the drawdowns now got only half the depths compared to the previous method. They look less dangerous, since unlike the other two methods, the square root method always keeps the same distance from the account balance to the maximum expected drawdown. This is the investment method you would normally prefer.

The square root rule

Often traders try to achieve a steady income through their activities. This means they must regularly withdraw a portion of their profits from the account. The square root rule must be applied here, too. If your capital has doubled, you can only pay out about 60% of the profit. The remaining 40% must remain on the account as a buffer to compensate future drawdowns.

Example: You start with a capital of $10000 and want to withdraw profit every time the system has won $3000. The first payout is therefore for the account balance of $13,000. Your investment grew by the growth factor 1.3; the square root of 1.3 is 1.14. So you have to keep $11,400 in the account and can pay $1600 to yourself. - A little later your system has collected another $3000. The account is now $14,400. The growth factor (disregarding the amounts taken) is **1+(3000+3000)/10000 = 1.6**. The square root of 1.6 is 1.265. $12,650 must remain in the account, $1750 are for you.

What if you do not want to withdraw the maximum amount, but reinvest the rest? Here are some simple formulas to estimate how much you can take and what remains to reinvest. In the following formulas, ***C*** is your starting capital,

P is the accumulated gain of the strategy or portfolio component, ***W*** is the sum of your withdrawals, and ***f*** is the square root of the growth factor, $\sqrt{1+\frac{P}{C}}$.

Balance on the account: ***C + P - W***
Minimum amount in the account: ***C f***
Available for removal: ***C + P - W - C f***
Available for reinvesting: ***C f - W/f***

Example: You start with a capital of $10,000 and win $3000. Your account balance now is at $13,000. How much can you increase your margin by the square root method after withdrawing $500 from the account?

Previous margin: **0.5 * OptimalF * $10000**
Account after profit: ***C + P-W*** **= $13,000**
Growth factor: ***1+P/C*** **= 1.3**, its square root ***f*** **= 1.14**
Account minimum: ***C f*** **= $10,000 * 1.14 = $11,400**
Maximum payout: ***C + P-W-C f*** **= $13,000-$11,400 = $1600**
Investment after $500 Payout: ***C f - W/f*** **= $11400-$500/1.14 = $10,960**
Margin for the next trade: **0.5 * OptimalF * $10,960**

So after $500 withdrawal from $3000 profit, you can increase your investment only from $10,000 to $10,960, not to $12,500 as you may have expected. Consider the difference as a tax that you pay to the god of statistics. Unfortunately you have to pay a real tax for this, too...
.

4 portfolio system tricks

► Combine many assets and algorithms in your strategy. The return may be lower than with the best single algorithm, but the system is more robust through diversification.

► Strategy parameters must be trained individually for any asset. It is rare that multiple assets share the same parameters.

► Optimize the capital allocation to strategy components, for instance by calculating **OptimalF** factors.

► With high leverage systems, increase your investment only proportionally to the square root of the capital growth.

6 Candles to kill for

Bob: "Quick, close the door! Maybe I've been shadowed."
Alice: "What happened?"
Bob: "Someone just revealed the ultimate trading secret to me."
Alice: "Again?"
Bob: "Yes, but this time really. Forget indicators, lowpass filters and all this stuff. It's price action trading. I need you to program this immediately. And talk to no one!"
Alice: "Price action? What's that?"
Bob: "There are formations of candles that show the initiate exactly where the price will go. You trade just when the price candle pattern is right. You compare the open, high, low and close of the last candle with the open, high, low and close of the previous candles - that's the pattern. It's all you need. The Japanese knew that already 300 years ago."
Alice: "Are those the patterns with the funny names like "Three Black Crows"? "
Bob: "You got it. Of course that was long ago and for the Japanese rice market only. But I know a guy, my old pal Bert who's working at McDuck Capital. He discovered new candle patterns for the forex market. He said it's like a slot machine. The pattern appears and cash comes out. Bert got a mad bonus and McDuck is since then trading price action with his patterns."
Alice: "So you want me to write a script that looks for those patterns and then triggers a trade signal? Should not be a problem. "
Bob: "Well, there is a problem. I don't know the patterns. I only know that they are made of three daily candles. "
Alice: "You don't know of which candles?"
Bob: "Not directly. Bert said he had to kill me when he told me that. McDuck is very serious in that matter."
Alice: "Hm."
Bob: "Can't you find out the patterns yourself?"
Alice: "If a guy at McDuck found them, I suppose I can find them too. But why do they work at all? I mean, why should a price move be triggered by a certain candle pattern?"
Bob: "No idea. But this method worked for the Japanese rice market. Maybe there are some big traders that wake up in the morning, look at the last 3 day candles, and if they like what they see they buy or sell. "

Alice: "If this establishes a pattern, I can apply a machine learning function. It goes through historic prices and checks which candle patterns usually precede a price movement up or down."
Bob: "Will that be expensive?"
Alice: "The search of candle patterns? No. Still, I'm afraid I'll have to charge more than last time."
Bob: "Why is that? "
Alice: "Risk surcharge. I might get killed when programming this script."

Detecting patterns

Alice uses Zorro's **advise** function for finding a system in candle patterns, and using the most profitable 3-candle patterns for a trade signal. This is her script (**Alice4a**, fee incl. risk surcharge: $8000):

```
function run()
{
   set(RULES+TESTNOW+PLOTNOW+LOGFILE);
   StartDate = 2010;
   EndDate = 2018;
   BarPeriod = 1440;
   BarZone = WET;
   Weekend = 1;
   LookBack = 3;

   WFOPeriod = 2000;
   DataHorizon = 3;

   if(Train) Hedge = 2;
   LifeTime = 3;
   Stop = 500*PIP;
   MaxLong = MaxShort = -1;

   if(adviseLong(PATTERN+2+RETURNS,0,
      priceHigh(2),priceLow(2),priceClose(2),
      priceHigh(1),priceLow(1),priceClose(1),
      priceHigh(1),priceLow(1),priceClose(1),
      priceHigh(0),priceLow(0),priceClose(0)) > 50)
      enterLong();

   if(adviseShort() > 50)
      enterShort();
}
```

Many lines in this code should be already familiar, but there are also some new concepts. Let's start with the most important one, the machine learning algorithm inside the **adviseLong** function. This looks like a strange entry condition for a long trade:

```
if(adviseLong(PATTERN+2+RETURNS,0,
   priceHigh(2),priceLow(2),priceClose(2),
   priceHigh(1),priceLow(1),priceClose(1),
   priceHigh(1),priceLow(1),priceClose(1),
   priceHigh(0),priceLow(0),priceClose(0)) > 50)
   enterLong();
```

AdviseLong and **adviseShort** are general functions for machine learning. Their parameters are the learning method, the training target, and the signals from which this target is to be trained. As a method, Alice has specified **PATTERN+2+RETURNS**, which will activate the pattern analyzing mechanism for two separate patterns. The **RETURNS** flag means that the result of the next trade will be the training target. And the signals are the high, low, and close price of the last three candles. If **adviseLong** returns a value greater than **50**, a long position is opened. But when will that happen?

When training, the **adviseLong** function always returns **100**. This means that the **> 50** condition is always fulfilled and the trade is always executed. This is necessary since its result serves as a training target. The **adviseLong** function stores a "snapshot" of the signal parameters in an internal list – in this case here, the 12 signals with high, low and close prices of the last 3 candles. Then it waits for the result of the trade and stores its gain or loss along with the signals. At the end of the WFO cycle, Zorro has a long list of candle prices and associated trade results for each bar.

The signals are then classified into patterns. Alice has put a **+2** to the **PATTERN** parameter. This tells Zorro's pattern analyzer to split the signals into two equal groups. Each got 6 of the 12 signals. The first group contains the prices of the first two candles of the 3-candle sequence:

```
priceHigh(2),priceLow(2),priceClose(2),
priceHigh(1),priceLow(1),priceClose(1)
```

And the second group contains the prices of the last two candles:

```
priceHigh(1),priceLow(1),priceClose(1),
priceHigh(0),priceLow(0),priceClose(0)
```

The middle candle - **priceHigh(1), priceLow(1), priceClose(1)** – belongs to both groups. Thus it appears twice in the list. The open price of a candle is not used in the signals because currencies are traded 24 hours a day, so the close of a daily bar is normally identical to the open of the next bar. Using the open price would emphasize outliers and weekend patterns where open might differ. This is not desired.

Within every signal group, Zorro now compares every signal with every other signal. This generates a huge set of greater, smaller, or equal results. This set of comparison results is a pattern. It does normally not matter if **priceHigh(2)** is far smaller or only a bit smaller than **priceHigh(1)** - the resulting pattern is the same (although there's also a 'fuzzy' comparison available where the difference matters).

The patterns of the two groups are now glued together to form a single pattern. It contains all price comparisons within the first and the second and within the second and the third candle, but not how the first candle compares with the third. Bert had told Bob that it's best for price action trading to compare only adjacent candles - therefore the two independent pattern groups. If Alice had looked for 4-candle-patterns, she had used three groups.

Aside from the grouping, Zorro makes no assumptions of the signals and their relations, such as that the high is always higher than the low. Therefore, in stead of candle patterns any other set of signals or indicators could also be used for the **advise** function and compared with the **PATTERN** method, or with other machine learning functions like decision trees, perceptrons, or neural networks from the R statistics package. If you're interested in this, visit the blog http://financial-hacker.com from time to time.

Back to Alice's script. After the pattern was classified, Zorro checks how often it appears in training data set, and sums up all its profits or losses. If a pattern appears often and with a profit, it is considered a profitable pattern. Zorro removes all unprofitable or insignificant patterns from the list - patterns that don't have a positive profit sum or appear less than 4 times. From the remaining patterns, a pattern finding function is generated and stored in the **Alice7a_EURUSD.c** script in the **Data** folder. Such a machine generated pattern finding function can look like this:

```
int EURUSD_L(float* sig)
{
  if(sig[1]<sig[2] && eqF(sig[2]-sig[4]) && sig[4]<sig[0] &&
sig[0]<sig[5] && sig[5]<sig[3]
```

```
    && sig[10]<sig[11] && sig[11]<sig[7] && sig[7]<sig[8] &&
sig[8]<sig[9] && sig[9]<sig[6])
      return 19;
  if(sig[4]<sig[1] && sig[1]<sig[2] && sig[2]<sig[5] &&
sig[5]<sig[3] && sig[3]<sig[0]
    && eqF(sig[8]-sig[10]) && sig[10]<sig[6] && sig[6]<sig[11] &&
sig[11]<sig[9])
      return 70;
  if(sig[1]<sig[4] && eqF(sig[4]-sig[5]) && sig[5]<sig[2] &&
sig[2]<sig[3] && sig[3]<sig[0]
    && sig[10]<sig[7] && eqF(sig[7]-sig[8]) && sig[6]<sig[11] &&
sig[11]<sig[9])
      return 74;
  if(sig[1]<sig[4] && sig[4]<sig[5] && sig[2]<sig[0] &&
sig[0]<sig[3] && sig[7]<sig[8]
    && eqF(sig[8]-sig[10]) && sig[10]<sig[11] && sig[11]<sig[9] &&
sig[9]<sig[6])
      return 43;
  if(sig[1]<sig[2] && eqF(sig[2]-sig[4]) && sig[4]<sig[5] &&
sig[5]<sig[3] && sig[3]<sig[0]
    && sig[10]<sig[7] && sig[7]<sig[8] && sig[8]<sig[6] &&
sig[6]<sig[11] && sig[11]<sig[9])
      return 68;
  ....
  return 0;
}
```

The functions in the code get their names from their asset, their algo, and whether they are used for long or short trades. This way many different functions can be stored in the same .c file. The function here has the name **EURUSD_L**. The list of signals is passed to the function as a float array named **sig**. The **float** type is a variable type similar to **var**, but has lower precision and thus consumes less memory. **sig[0]** is the first signal passed to the **advise** function - in this case, it's **priceHigh(2)**. **sig[1]** is the second signal - **priceLow(2)** - and so on. The signals are used inside the function just like the elements of a series.

We can see that the function contains many **if()** conditions with many comparisons of signals with other signals. Any such **if()** condition represents a pattern. The comparisons are linked with **&&**, which is the same as the **and** operator. So the **if()** condition is true only when all its comparisons are true. In this case a certain value, like **19** in the first **if()** condition in the above example, is returned by the function. If none of the **if()** conditions is true, 0

is returned, meaning that no pattern is found. The returned value is the pattern's score - its 'information ratio' multiplied with 100. The higher the information ratio, the more predictive is the pattern.

Alice compared the returned value with **50**, meaning that a long trade is entered for any pattern match with a score above 50.

Short trading just works the same way:

```
if(adviseShort() > 50),
  enterShort();
```

AdviseLong and **adviseShort** are identical functions, but each carries out its own pattern analysis. In this way long and short trades can be based on different patterns. The **adviseShort** call here has no parameters. This tells Zorro to simply use the methods and signals from the last **advise** call, which was the preceding **adviseLong** call. This saves some paperwork in the script. The learning target here is different of course, because the result of the following short trade is now evaluated.

Machine learning

Some prerequisites for learning candle patterns haven't been discussed yet. Let's look at the rest of the code:

```
set(RULES+TESTNOW+PLOTNOW+LOGFILE);
```

The **RULES** flag is like the **PARAMETERS** flag, but the system won't train parameters here, it's learning trading rules and applies them in test or trade mode. This flag is always required for the **advise** function.

```
StartDate = 2010;
BarPeriod = 1440;
BarZone = WET;
Weekend = 1;
LookBack = 3;
```

For detecting daily patterns, the bar period is 1440 minutes, or 24 hours. Normally, daily bars begin and end at UTC midnight. But for price patterns the time zone of the bars is critical. Alice has to catch a time with low vola-

tility. A good time for low EUR/USD volatility is midnight in Western Europe. **BarZone** determines the time zone of a daily bar; **WET** is the Western European Time, the time zone of London, considering daylight saving time.

The **Weekend** variable determines how the simulation deals with weekend bars. Normally, no bar is allowed to start or end within a weekend. This means that for daily bars, the bar starting Friday 00:00 midnight would end Monday 00:00 midnight. This is not desired here because this bar would then contain prices from Friday as well as from Sunday evening, and spoil the candle pattern. Thus, **Weekend = 1** enforces the Friday bar to end Saturday 00:00 midnight, although due to the weekend no trades can be entered on that bar. The week then consists of 6 instead of 5 daily bars.

Because the strategy needs only the last 3 candles for trade decisions, we can set the **LookBack** period from its default 80 down to 3 bars. This gives us three more months for training and testing.

```
WFOPeriod = 2000;
DataHorizon = 3;
```

WFO or some other out-of-sample test method is mandatory for this type of strategy. All machine learning systems tend to overfitting, so any in-sample result from price patterns, decision trees, perceptrons, or neural networks would be too optimistic and thus meaningless. **WFOPeriod** is an alternative to the familiar **NumWFOCycles**. It gives the number of bars per cycle. Zorro will then calculate the number of cycles from it. 2000 bars are equivalent to 8 years (a year has about 250 trading days). That's a very long time for a WFO cycle - and, as we will see later, also a fundamental weakness of this strategy. But finding profitable candle patterns requires a large number of candles. Much less than 2000 would not do. Much more would neither, because one can imagine that the market changes substantially in a decade, meaning that patterns get outdated and new patterns emerge. So the 2000 bars are a compromise – possibly a bad one.

The number of bars for training could theoretically also be increased with a method named **Oversampling**. You can read about it in the Zorro manual. Unfortunately, oversampling would work for hour bars, but not for daily bars that are bound to a certain time zone and cannot be arbitrarily shifted.

The **DataHorizon** variable inserts a gap between WFO cycles. Without this gap, peeking bias would creep into the backtest. That's one of the traps of machine learning functions, since they normally use a future value as training

target – in this case, the future result of a trade. Thus, the training overlaps the begin of the test period a bit. In this case, by 3 bars since that is the maximum lifetime of a trade. **DataHorizon** prevents the overlapping. It does not matter much here since 3 is small compared to the 300 bars of a test cycle (15% of 2000), but in other configurations it can matter a lot.

The next code part behaves different in training and in test or trade mode:

```
if(Train) Hedge = 2;
```

Train is true in **[Train]** mode. In this mode we want to determine the return of a trade that follows a certain pattern. **Hedge** is set to 2, which allows long and short positions to be open at the same time. This is required for training the patterns, otherwise the short trade after **adviseShort** would immediately close the long positions that was just opened after **adviseLong**, and thus assign a wrong return value to its candle pattern. **Hedge** is not set in test and trade mode since it makes sense that positions are closed when opposite patterns appear.

```
LifeTime = 3;
Stop = 500*PIP;
```

LifeTime here sets the duration of a trade to 3 bars maximum, considering that its result prediction is also based on the last 3 bars. So if a trade is not closed by an opposite pattern, it is closed after 3 trading days. Additionally, Alice has set a stop loss, however in a large distance of 500 pips, so that it only limits the loss in extreme cases.

But where's the trading filter? Because it is unknown under which market conditions patterns emerge or not, Alice does not know how to use a filter for this strategy.

Make sure that **EUR/USD** is selected, then click **[Train]**. Zorro will need only a few seconds for running through the WFO cycles and finding the profitable long or short patterns in every cycle. There are about 30 patterns per cycle, whose C codes are stored in **.c** files in the **Data** folder. After training, a walk-forward test is performed immediately due to the **TESTNOW** flag. The result:

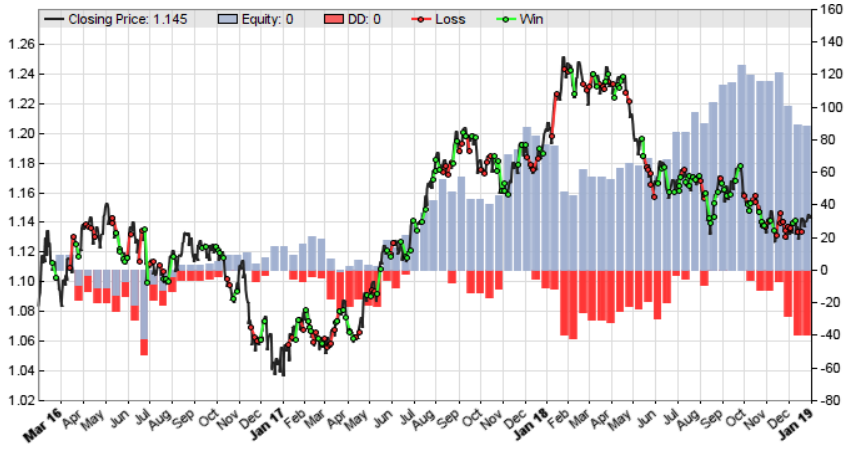


Fig. 36 – Trading with three-days candle patterns

The machine learning algorithm with daily candle patterns seems to give us a more or less rising equity curve and symmetric results in long and short trading. This was apparently the top secret algorithm that Bert presented to his employer. But was his mad bonus justified? Or was his employer just fooled by randomness? After all, the win rate is only slightly above 50%. Can Alice trust this result? Her gut feeling says: No.

Reality check

Alice has used the walk-forward mechanism to rule out that the positive result was caused by overfitting bias. There is, however, another effect that strongly influences trade results: simple randomness. Patterns of all kinds can also be found in arbitrary random curves. The question is whether the found patterns can produce such a wide spread of random results that the 40% profit have no significance at all. In this case, a slightly different price curve could as well result in a 40% loss. You don't want to trade such a system.

The dependence of a strategy on randomness can be determined relatively easily. You only need to test it with random curves instead of a real price curve. For this, Alice inserts the following line:

```
Detrend = SHUFFLE;
```

The **Detrend** variable eliminates trends from the trade results or from the price curve with various methods. **SHUFFLE** removes all short-term trends

by wildly shuffling all price differences. This way they still have the same statistical properties. The mean, the long-term trend and the volatility have not changed, and a trader would most likely not be able to distinguish such a shuffled price curve from a real one. But any relation between individual bars is completely destroyed. If there were inefficiencies in the price curve, they are now gone. This also applies to all possibly predictive patterns.

Add the **Detrend = SHUFFLE** line to the beginning of the run function, and then click **[Test]** repeatedly. Each time you should get a different result - sometimes positive, sometimes negative. Most results are worse than the result with the real price curve - but is this significant?

To find out, there is an obvious way: run many tests with many different random curves and see how the real result compares with all the random results. For not having to click a thousand times on **[Test]**, Alice has added and modified some lines in her script (**Alice4b.c**):

```
if(Test) {
   NumTotalCycles = 1000;
   if(TotalCycle > 1)
      Detrend = SHUFFLE;
}
```

In test mode, the variable **NumTotalCycles** is set to 1000. This variable specifies how often the entire simulation is to be repeated. It is very useful to generate a histogram or statistic of results from many simulations. **TotalCycle** is the number of the current simulation. **Detrend = SHUFFLE** is set after the first simulation to generate randomly shuffled prices. The first simulation still runs with the original price curve. Train mode needs not be repeated 1000 times since we've used WFO. The test is out of sample, so it does not matter whether the training was done with shuffled data or with the original price data.

```
set(RULES+PRELOAD);
```

We now cannot run anymore a test after training or plot a chart after any test. The **LOGFILE** flag has also disappeared because a test with no logging is faster. This matters when we're going to run the test a thousand times. But we must set the **PRELOAD** flag. It causes the price curve to be loaded at the start of any simulation cycle, which is necessary because we have to shuffle it every time.

There is also a change to the **adviseLong** call:

```
if(adviseLong(PATTERN+FAST+2+RETURNS,0,...
```

The added **FAST** flag uses a faster method to store patterns. It does not produce rules in C code, but encodes patterns as character sequences. As a result, the candle patterns can be detected twice as fast, which is not unimportant when the simulation is repeated 1000 times. The **FAST** flag also allows more complex patterns and consumes less memory, so it should always be used for pattern detection unless rules in C code are needed for some reason, like exporting them to another platform.

Aside from those modifications, there's a new function in the script:

```
function evaluate()
{
   static var OriginalProfit, Probability;
   if(TotalCycle == 1) {
      OriginalProfit = Balance;
      Probability = 0;
   } else {
      plotHistogram("Random",Balance,10,1,RED);
      if(Balance > OriginalProfit)
         Probability += 100./NumTotalCycles;
   }
   if(TotalCycle == NumTotalCycles) {
      plotHistogram("Original",OriginalProfit,10,45,BLACK);
      printf("\n-----------------------------------------");
      printf("\nP-Value %.1f%%",Probability);
      if(Probability <= 1)
         printf("\nResult is significant");
      else if(Probability <= 5)
         printf("\nResult is possibly significant");
      else
         printf("\nResult is statistically insignificant");
      printf("\n-----------------------------------------");
   }
}
```

This function produces the reality check histogram. Let's look into the details.

Bell curve and p-value

The function **evaluate** is a special function, just as **run**. It is called automatically at the end of every simulation, and is used to evaluate or store results. Inside this function, first two static variables are declared:

```
static var OriginalProfit, Probability;
```

The addition **static** has the effect that the current value of the variable is preserved even if the function is exited. The variables are set up in the following code:

```
if(TotalCycle == 1) {
   OriginalProfit = Balance;
   Probability = 0;
} ...
```

Balance is the sum of the returns from all closed trades, and therefore the total profit of the simulation (it is not the same as **ProfitClosed** in the previous chapter: **ProfitClosed** refers to the current portfolio component only, **Balance** to the entire simulation with all components.) The variable **OriginalProfit** is set to the end balance of the first simulation **(TotalCycle == 1)**, which used the original, unshuffled price curve. **Probability** is initialized to 0. Static variables need to be explicitly initialized because they would otherwise keep their values from the last run.

The **else** clause produces a histogram of the profit distribution and counts the frequency of balances that are higher than the original profit:

```
... else {
   plotHistogram("Random",Balance,10,1,RED);
   if(Balance > OriginalProfit)
      Probability += 100./NumTotalCycles;
}
```

A histogram consists of individual bars, each representing the number of occurrencies of a particular value – here, of the balance. For this purpose, the **plotHistogram** function increases an individual counter for each balance value that falls in a certain range. The final histogram displays the counters as red bars. For **plotHistogram** the **profile.c** module must be included.

If the balance of the current simulation is higher than the original profit, an amount of **100./NumTotalCycles** – that's **0.1** in this case – is added to the

Probability variable. Here we have to take care of the peculiarities of a programming language. **NumTotalCycles** is a variable of type **int**. If Alice had written

```
Probability += 100/NumTotalCycles;
```

then **Probability** would never increase, because the integer result of 100/1000 is 0. An operation of two **int** always results in an **int**. That's why Alice has added the decimal point to **100**, indicating that it's a **var**. An operation of a **var** with an **int** results in a **var** and produces the correct value 0.1.

We want to see what histogram is produced by our reality check. To do this, first click **[Train]** to learn the patterns, then click **[Test]** to apply them to 1000 different random curves. Now you have to wait about 10 minutes until the 1000 simulations are finished. The result should look like this:

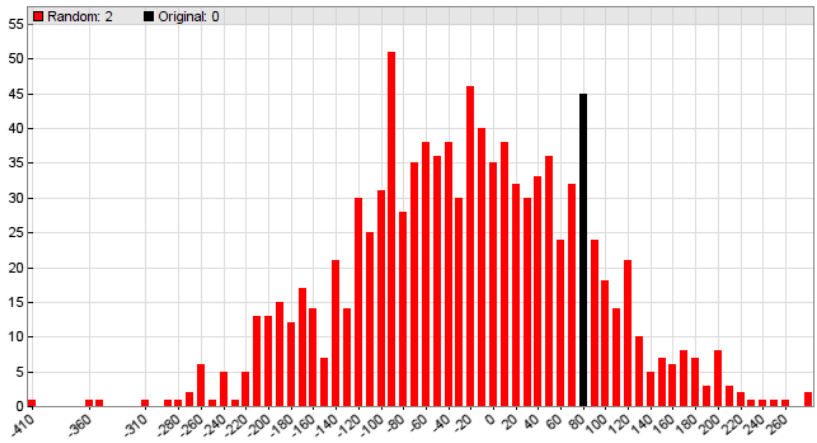


Fig. 37 – Reality check histogram

This is a "bell curve" - the Gaussian normal distribution. The height of the bars indicates how often the assigned balance occurred during the tests. The bell appears a little frayed, since 1000 samples are not enough to get a smooth curve. But we can already see that the top of the bell - the area of the highest bars - is at a profit of slightly below 0. This was expected, since trades with random patterns generate an average profit of zero dollars. The bell top is somewhat offset to negative values because transaction costs such as spread and commission reduce the average profit.

The most interesting thing to us is, of course, the black bar that indicates the profit from the real price curve. It is plotted at the end of all simulation cycles, together with the reality check result:

```
if(TotalCycle == NumTotalCycles) {
   plotHistogram("Original",OriginalProfit,10,45,BLACK);
   printf("\n-----------------------------------------");
   printf("\nP-Value %.1f%%",Probability);
   if(Probability <= 1)
      printf("\nResult is significant");
   else if(Probability <= 5)
      printf("\nResult is possibly significant");
   else
      printf("\nResult is statistically insignificant");
   printf("\n-----------------------------------------");
}
```

The condition **if(TotalCycle == NumTotalCycles)** is fulfilled at the end of the last simulation run. At this point, a black bar with the original balance and height 45 is plotted in front of the histogram. The variable **Probability** was incremented by 0.1 in all simulations that resulted in a balance higher than the original profit. If that happened never, **Probability** had still the value 0 at the end. If it happened always except for the first simulation, **Probability** ended up at 99.9. Thus, **Probability** indicates the significance of the original result in comparison to the simulation. It is the so-called **p-value** of the reality check. The smaller the p-value, the higher the significance. Statisticians see a result as significant if its p-value is below 1%. If the p-value is less than 5%, the result is considered "potentially significant", with all higher p-values being "not significant".

Alice has no choice but to report a result like this to Bob:

```
-----------------------------------------
P-Value 16.4%
Result is statistically insignificant
-----------------------------------------
```

This is a strategy that would not work in live trading because the positive test result, although achieved with WFO, was random. Bert's bonus was unjustified and McDuck's further fate uncertain. Candle pattern trading, also called 'Price Action', apparently does not work – at least not with EUR/USD and with daily patterns of 3 candles. The problem is not the used algorithm, which will find all patterns. The problem is that candle patterns have short lifetime and weak predictive power. Predictive patterns do not survive long enough

for being safely detected within daily bars. This could be different with hourly bars. The reader is invited to find out.

The reality check described here can determine that a strategy won't work. But the opposite is not true: it cannot with the same certainty determine that it will work. The check will fail with overfitted strategies. If an algorithm and its parameters - such as trade duration or number of WFO cycles – are consciously or unconsciously selected in a way to make the result as positive as possible, it can also pass this reality check. There is a more general procedure, **White's Reality Check**, that can quantify such effects and even eliminate them from the results. This method is well described in Aronson's book listed in the appendix; code for it can also be found on **www.financial-hacker-com**. Unfortunately, White's Reality Check is great in theory, but difficult to use in the real development process of a trading strategy. It requires recording and testing all strategy variants that were discarded during development by any decision influenced by a test result - for example, of how many candles the pattern shall consist, or of how many groups. It is cumbersome to keep records of all discarded variants and their test results. The best way to achieve this is with strategies that are completely machine generated and unaffected by result-based decisions, such as in Aronson's indicator study, or in a similar study of trend indicators on the Financial Hacker blog.

5 advices for price action trading

► Out-of-sample testing is is mandatory for machine learning strategies.

► Use a **Reality Check** for determining the effect of randomness on your test result.

► There are many methods for detecting patterns in data. Read the manual of your trading software to make yourself familiar with the available functions. An **advise** call generates trade rules in C code; **Detrend = SHUFFLE** tests a strategy with a random price curve; **NumTotalCycles** can generate statistics from multiple simulation runs; **plotHistogram** plots histograms.

► Learn peculiarities of the used programming language for finding shortcuts and avoiding traps that would otherwise require a long debugging session. For instance, **static** variables keep their value between function calls; an arithmetic operation of **int** with **int** produces an **int**; an operation of **var** with **int** however produces a **var**.

► Last but not least, don't be disappointed if a strategy or a whole trading method turns out unprofitable. Even this is a hard-won, useful result. And it is worth hard money - the money you would otherwise have lost while trading the system.

7 In the 5th dimension

Bob: "I've done everything wrong so far."
Alice: "Everything?"
Bob: "I finally found someone who explained trading to me. So far I had no idea. Forex - that's beginner's stuff. Real trading is options trading."
Alice: "How does that work?"
Bob: "Well, options, that's trading in five dimensions. I have not fully understood it yet, but that's ok, that's why I pay you. So I want a strategy that trades with options. Just buy the right option at the right moment. No, I meant of course, sell it. Options have the seller advantage."
Alice: "I see."
Bob: "I habe been credibly informed that this way it's very easy to make a profit. In fact, you can hardly lose when you sell options. Or if you did, you you had programmed something wrong."
Alice: "So I'm supposed to program a strategy that sells the right option at the right moment and inevitably makes a profit because of the seller advantage."
Bob: "You got it. And make the strategy as simple as possible. Can you do that?"
Alice: "I have to buy a book on options first."

Options 101

An option is not a financial asset. It is a simple contract that gives its owner the right to buy (**call** option) or sell (**put** option) a certain amount (the **multiplier**, usually 100) of a stock, a future, or some other asset (the **underlying**) at a fixed price (the **strike**) at or before a fixed date (the **expiry**). Better read the previous sentence twice. If you sell short (**write**) an option, you're taking the other side of the contract. For this you collect a **premium** from the buyer. So you can enter a long call, or a long put, or a short call, or a short put. All this with a large range of strikes and expiry dates. That's why Bob mentioned five dimensions, since an option position can be described with five parameters: type, premium, strike, underlying price (**spot**), and expiry. It is also

usual to trade not only single options, but combinations (**combos**) of different options types. Thus there are innumerable possibilities for options strategies.

The **premium** is normally far less than the price of the underlying amount. This means a high leverage for buying options positions. Major option markets are usually liquid, so you can anytime buy, write, or sell back an option with any reasonable strike and expiry. If the underlying price (the **spot** price) of a call option lies above the strike, the option is **in the money**; otherwise it's **out of the money**. The opposite is true for put options. Expiring in the money is good for the buyer, expiring out of the money is good for the seller. Options in the money can be **exercised**, meaning that they are then exchanged for the underlying at the strike price. The difference of spot and strike is the buyer's profit and the seller's loss. **American style** options can be exercised anytime, **European style** options only at the expiry date.

Out-of-the-money options can not be exercised, at least not at a profit. But they are not worthless. They have still a chance to move into the money before expiration. The value of an option depends on that chance and can be calculated for European options from spot price, strike, expiry, riskless yield rate, dividend rate, and underlying volatility with the famous **Black-Scholes formula**. This value is the basis of the option premium. The real premium might deviate dependent on supply, demand, and assumed price trend of the underlying. This also works the other way around: Options premiums can be used to calculate the market's belief in future prices of the underlying. Zorro has a function for that (**contractCPD**).

Due to the premium, options can still produce a profit to their seller even if the underlying moves a bit in the wrong direction. That's the **seller advantage**. Normally, such advantage would quickly disappear due to a seller overhang that reduces the premium. But options are often bought as an insurance against unwanted price moves of the underlying. So there's a balance of sellers and buyers that keeps the seller advantage alive. At least most of the time.

There's more to options, but that would exceed the scope of this book. For all the details, like implied volatility, the greeks, the volatility smile, iron condors, butterflys, and similar stuff, do it like Alice and get a book about options trading. But since Alice's task is writing a plain selling strategy, the basics described here are sufficient.

The seller's advantage

This is Alice's options selling script (**Alice5a**, agreed fee: $10,000):

```
#include <contract.c>
#define CALLPREMIUM  5.00
#define PUTPREMIUM   10.00
#define WEEKS     6

void run()
{
   set(PLOTNOW|LOGFILE);
   BarPeriod = 1440;
   History = ".t8";

   assetList("AssetsIB");
   asset("SPY");
   Multiplier = 100;

   vars Prices = series(priceClose());
   if(!contractUpdate(Asset,0,CALL|PUT)) return;

   var Vola = abs(ROC(Prices,WEEKS*5));
   if(NumOpenShort || Vola > 10) return;

   if(combo(
      contractFind(CALL,WEEKS*7,CALLPREMIUM,2),1,
      contractFind(PUT,WEEKS*7,PUTPREMIUM,2),1,
      0,0,0,0))
   {
      MarginCost = 0.15*Prices[0]/2;
      Commission = 1.0/Multiplier;
      enterShort(comboLeg(1));
      enterShort(comboLeg(2));
   }
}
```

We can see some new concepts. Options are normally not offered by forex/CFD specialized brokers, so Bob has opened an account with a stock broker that offers all sorts of financial instruments. The file **AssetsIB.csv** is a list of selected assets by this broker, and Alice has to explicitly load it with the **assetList** function. Otherwise only currencies and CFDs from Zorro's scroll box had been available in the script.

```
assetList("AssetsIB");
asset("SPY");
History = ".t8";
```

SPY is an **ETF** (**E**xchange **T**raded **F**und) that follows the main US stock index, the S&P 500 index. Alice is not trading it directly, but she's trading options on SPY. The predefined string **History** can be set to the end of a file name for telling Zorro from which file type to load the historical prices and options. It must be set before loading the data with the **asset** call. Here, the setting **".t8"** loads both from **SPY.t8**, a file with historical options data that Bob has purchased from a data vendor. Unlike forex data, options data is usually not free. Depending on length and resolution it can be a couple hundred dollars. Options data selling is apparently even more lucrative than options selling.

For testing purposes, a free SPY options data set **SPYa.t8** is available on the Zorro website. For using it, put it in the History folder and set **History = "a.t8"**. But be aware that it is artificially generated with the Black Scholes Algorithm, and thus does not reflect market sentiment. It produces different results with some strategies, like this one.

```
Multiplier = 100;
```

Multiplier gives the number of underlying shares per option. For stocks and ETFs, it's usually 100. This variable must be set for any asset before options can be traded. Option values and premiums are also usually given per underlying unit and must be multiplied with the **Multiplier**.

```
vars Prices = series(priceClose());
if(!contractUpdate(Asset,0,CALL|PUT)) return;
```

For any asset, thousands of options are normally available with all possible combinations of type, strike, and expiry. The **contractUpdate** function loads all of them, either from the broker, or out of the historical data. If no options are available, for instance due to a holiday, the function returns 0 and the script also returns and skips the rest of the code. Alice has generated the **Prices** series in the line before that because **series** calls must not be skipped.

```
var Vola = abs(ROC(Prices,WEEKS*5));
if(NumOpenShort || Vola > 10) return;
```

This is Alice's filter condition. If option positions are already open, or if the volatility is higher than 10%, the script returns and does not trade. We will

see later why her strategy works better in less volatile market situations. For measuring volatility, Alice checks through the **ROC** indicator by how much percent the price has changed in the last 6 weeks. The magnitude (**abs**) is taken because the direction of the change does not matter. The 6 weeks are identical to the life time of the contracts and have been defined at the begin of the strategy:

```
#define WEEKS  6
```

#define statements are useful for managing parameters or code variants at the begin of the script. Alice has the habit to write all #define tokens in capitals for distinguishing them from variables.

The next lines are the core of the strategy:

```
if(combo(
   contractFind(CALL,WEEKS*7,CALLPREMIUM,2),1,
   contractFind(PUT,WEEKS*7,PUTPREMIUM,2),1,
   0,0,0,0))
{
…
```

contractFind looks through the option chain – the huge list of all option contracts – and returns the option closest to the given expiry and premium. If none is found, it returns 0. Alice looks first for a call with 6 weeks expiration and $5 premium, then for a put with $10 premium. We'll come later to the reasoning of selling a put and a call at the same time, and why the put premium is twice as high as the call premium. Note that the expiration in days was **WEEKS*7**, while the number of bars for the volatility, which Alice wanted to sample from a similar time period, was **WEEKS*5**. 5 bars are equivalent to 7 days since there are no bars at the weekend. If both contracts are found, the **combo** function bundles both options together to an option combination and returns nonzero. In that case, margin and commission are calculated and the options are sold:

```
MarginCost = 0.15*Prices[0]/2;
Commission = 1.0/Multiplier;
enterShort(comboLeg(1));
enterShort(comboLeg(2));
```

Zorro's automatic margin and commission calculation works only for normal assets, not for option margins that are different with any broker and any option combination. They must be calculated per script and stored in

the **MarginCost** and **Commission** variables before entering the trade. For simultaneously selling a call and a put, Bob's broker charges $1 commission per option, and requires a margin of 15% of the current underlying plus both premiums. The premium booked on the account is in fact a loan until expiration. Zorro considers this automatically by booking the premium not at opening, but only at closing the trade. 15% of the current price remain as initial margin cost. It is divided by two because cost is assigned to every option of the combination. The margin cost is already per underlying unit, the $1 commission is not and must therefore be divided by the multiplier. Zorro expects all costs as an amount per underlying unit.

The options are then sold with normal **enterShort** calls. The **comboLeg** function selects the option and returns the number of contracts – here, **1** – as first parameter to **enterShort**. Without calling **comboLeg**, or selecting the option otherwise, **enterShort** would sell the underlying instead of the option.

Now it's time to test the system. This is the result with 'real' SPY options history:

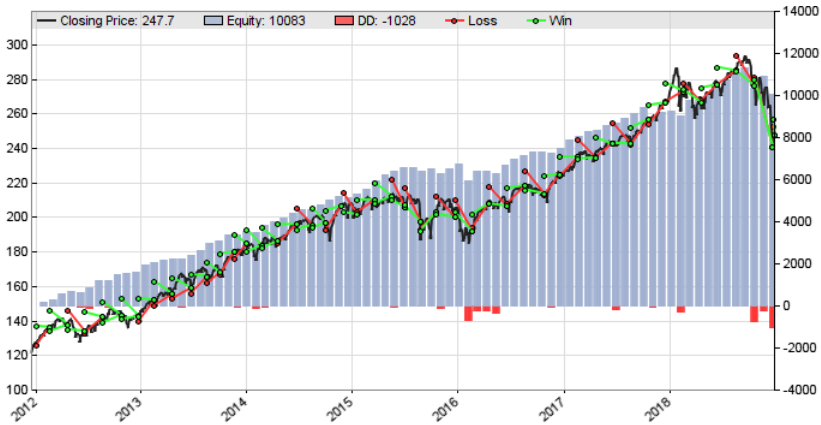


Fig. 38 – Selling puts and calls

The annual return is about 20%. With the artificial **SPYa.t8** history it would be higher, but that's unfortunately of no use for us. 20% do not sound exciting compared to the previous forex strategies, but this system produces profits steadily, well suited for a regular income. It only fails when the SPY price undergoes strong fluctuations, as in 2015 and 2018. That's the reason of Alice's volatility filter. Which is, by the way, rather simple in this script and can certainly be improved by the reader.

But how can selling a put and a call at the same time produce profit at all? Won't they cancel out each other?

Payoff diagrams

For the following test you'll need **R** and its **RQuantLib** package installed on your PC, and the path to the R terminal entered in **Zorro.ini**. Look up installation details in the Zorro manual under "R Bridge" and "Options & Futures". R is a popular language for data analysis and machine learning, and RQuantLib is a R package for option calculations.

Open the **Payoff** script in the editor, and edit the function **doCombo** so that it looks like this:

```
void doCombo() {
   contractAdd(-1,CALL,-5);
}
```

Click **[Test]**. If R and RQuantLib are correctly installed, you should see a diagram like this:

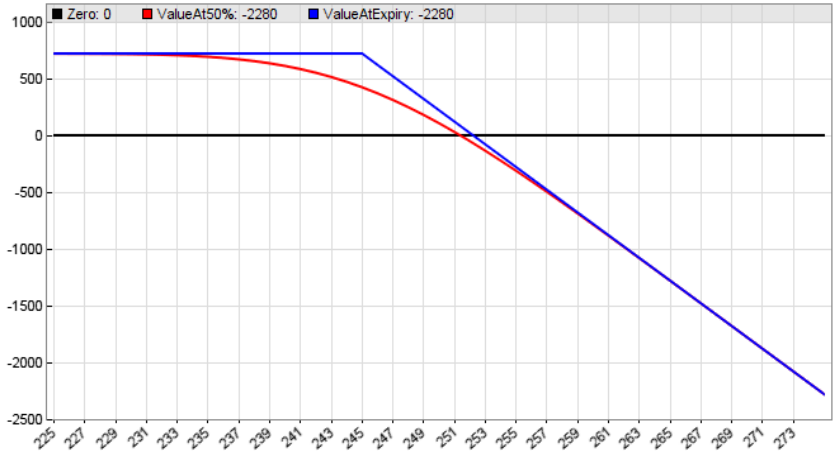


Fig. 39 – Call payoff diagram

This 'payoff diagram' shows the profit or loss of a call sold at strike $5 below a $250 spot price. Since the call is sold in the money, it earns an accordingly

high $7.20 premium. Considering the multiplier, $720 are booked on our account. The blue line in the diagram (we can ignore the red one) shows the outcome in dependence of the price of the underlying at expiration. If the price drops to $245 or below, the call expires out of the money and we keep the premium. If it stays at $250, we can see in the diagram that we still earn about $220. How so? The option is exercised in the money, and our broker sells to the option holder from our account 100 shares of the underlying at $245, like it or not. When we immediately cover the position, we lose $5 with any share. Fortunately, we collected $7.20 per share premium. Our total win is still $220, minus ask-bid spread and commission.

If the underlying price rises to $252.20 or above, we start losing. If it rises a lot, we lose a lot. There's theoretically no limit to our loss. That's the bad thing with selling a call.

Now suppose we sell a put at strike $10 above the underlying:

```
void doCombo() {
   contractAdd(-1,PUT,10);
}
```

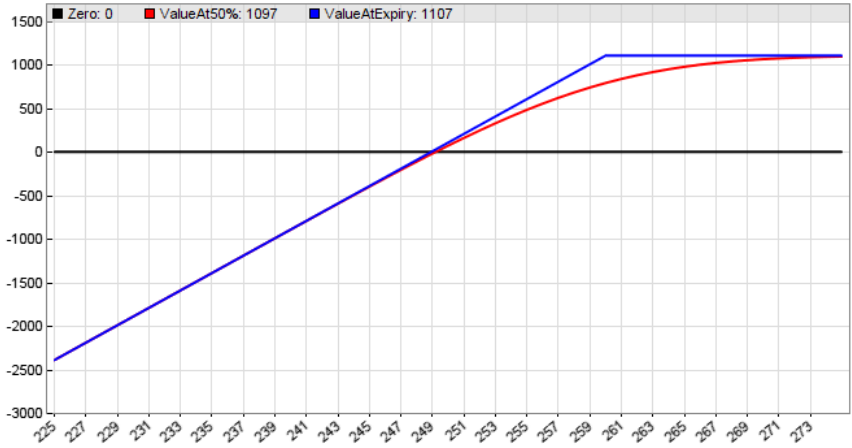


Fig. 40 – Put payoff diagram

The large strike distance to the spot price earns a fat premium of $11.07. So we collect $1107 and can keep it when the price rises to $260 or above. But if the price drops and the put expires in the money, our breakeven is at S249, barely below the price at selling. Anything lower would be bad.

There's another problem with a short put or call expiring in the money, even if we stay in the profit zone. With a put, the contract holder will enforce 100 shares at $260 to us. That means we must pay. Or rather, the broker will pay the margin with our money. Same happens with a short call, only we end up with a short position in the underlying. So we need enough cash in our account – in this case, $26,000 divided by leverage. It does not matter that we're going to sell the unwanted stock immediately. For avoiding margin calls or liquidation, this amount must permanently remain on our account, and reduces the annual return.

What happens if we do it like Alice and sell a put and a call?

```
void doCombo() {
   contractAdd(-1,CALL,-5);
   contractAdd(-1,PUT,10);
}
```

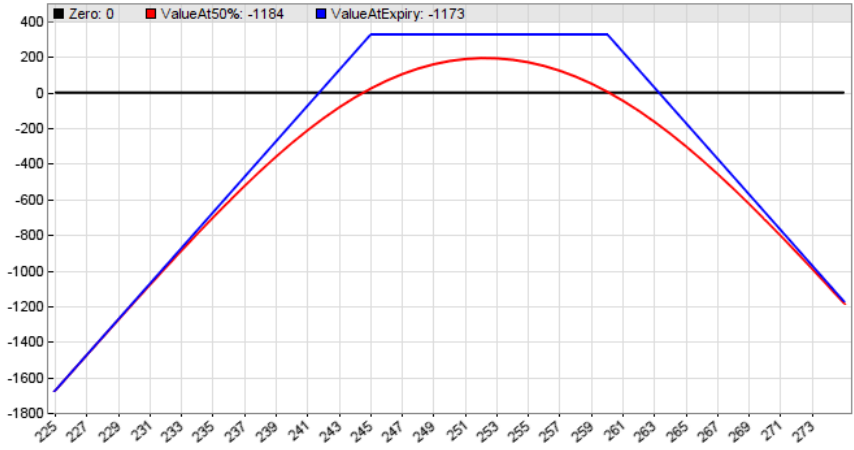


Fig. 41 – Put/call payoff diagram

At least one, if not both options will now with certainty expire in the money. This reduces the maximum possible profit. It's not the sum of both premiums, but only about $327. Even worse, we'll now lose when the price rises a lot, and also lose when it drops a lot. Seems like a bad deal, but there's one bright side. The breakeven points have wandered further away. We're still in the winning zone when the spot price stays between $242 and $263. Mind the asymmetry by the different strike distances. The initial $250 price may drop by $8, but rise by $13 for still achieving a profit. That's a good thing,

because stock prices are more likely to rise than to drop. At least most of the time.

Now let's look into the fate of a real put/call combination that Alice has sold. Open Alice5a_test.log and scroll down to the first trade:

```
[98: Tue 12-01-03 15:40] (127.50)
[SPY::SC9801] Write 1 Call 20120218 126.0 100@4.80 Unl 127.50
[SPY::SP9802] Write 1 Put 20120218 137.0 100@9.65 Unl 127.50
```

The above log is from a test with the real **SPY.t8** data. With the artificial data **SPYa.t8** you'll see different values, but let's continue as if you had the real thing. Alice has not looked for certain strike distances, but for premiums of $5 and $10. The **contractFind** function found the two options with closest premiums. It's a call in the money by $1.50 ($127.50 spot minus $126 strike), and a put in the money by $9.50. The total premium is $480 + $965 = $1445. Now scroll down 6 weeks to the expiration:

```
[SPY::SC9801] Expired ITM 1 Call 20120218 126.0 100@136.41: -562
[SPY::SP9802] Expired ITM 1 Put 20120218 137.0 100@136.41: +903
```

Both positions expired in the money (**ITM**), the first one at $562 loss, the second one at $903 win, resulting in $341 total profit. Let's verify it: The broker sold 100 SPY shares at $126 to the first holder and got from the second 100 shares at $137. The $1100 loss subtracted from the premium yields $345 end result. The remaining difference to the real $341 win is the broker's commission, about one dollar per transaction. We assume that Zorro has calculated it correctly.

Money management for options

In options systems, capital is not only needed for surviving drawdowns and covering margins, but also for possibly paying the underlying that's assigned when options get exercised. Thus, the money management will be a bit different than for leveraged forex systems. And Alice needs to consider some subtleties required for live trading. Because the system trades only every six weeks, it makes no sense to run it permanently. Bob will just start it every 6th Monday morning. It should then sell any previously assigned underlying position, enter new option positions, and quit.

Alice accordingly modified the script for live trading and reinvesting (**Alice5b**):

```
#define CALLPREMIUM  5.00
#define PUTPREMIUM   10.00
#define WEEKS     6
#define LEVERAGE  4
#define LIQUIDATE

void run()
{
   set(PLOTNOW|LOGFILE);
   StartDate = ifelse(Live,NOW,20120101);
   EndDate = 20181231;
   BarPeriod = 1440;
   History = ".t8";
   SaveMode = 0;
   Capital = slider(1,10000,0,20000,"Capital","");

   assetList("AssetsIB");
   asset("SPY");
   Multiplier = 100;
   setf(TradeMode,TR_GTC);

   contractSellUnderlying();
   vars Prices = series(priceClose());
   if(!contractUpdate(Asset,0,CALL|PUT)) return;

   var Vola = abs(ROC(Prices,WEEKS*5));
   if(NumOpenShort || Vola > 10) return;

   if(combo(
      contractFind(CALL,WEEKS*7,CALLPREMIUM,2),1,
      contractFind(PUT,WEEKS*7,PUTPREMIUM,2),1,
      0,0,0,0))
   {
      Commission = 1.0/Multiplier;
#ifdef LIQUIDATE
      MarginCost = 0.15*Prices[0]/2;
#else
      MarginCost = comboStrike(2)/LEVERAGE/2;
#endif
      var TotalCost = 2*Multiplier*MarginCost;
      Lots = (Capital+ProfitClosed)/TotalCost;
      if(Lots >= 1) {
         enterShort(Lots*comboLeg(1));
         enterShort(Lots*comboLeg(2));
```

```
    }
  }
  if(Live && !is(LOOKBACK)) quit("Ok!");
}
```

Alice's modifications in detail:

```
StartDate = ifelse(Live,NOW,20120101);
```

The **ifelse** function returns the second parameter if the first one is nonzero, otherwise the third. In live trading (**Live**), **NOW** is returned, meaning that trading shall start immediately after the lookback period without waiting for the end of the bar.

```
SaveMode = 0;
```

This is particular to Zorro: Alice switches off the mechanism that saves and resumes open trades between sessions. Otherwise there would be always messages about expired options when Bob starts the script.

```
Capital = slider(1,10000,0,20000,"Capital","");
```

Bob can now set up his investment with the slider.

```
setf(TradeMode,TR_GTC);
```

Since the strategy is not permanently running, it can not permanently monitor the trades. Therefore, Alice activates the Good-Til-Canceled (**GTC**) mode. In this mode, Zorro just sends the orders without waiting for execution. This makes sense because opening option combinations may take several minutes even at market hours.

```
contractSellUnderlying();
```

Bob doesn't want to keep the underlying that he got assigned when an option is exercised. Therefore, this function checks in live trading if a position of the current underlying is open. If so, it immediately sells it at market. Check out the source code of that function – it can be found in **contract.c**.

```
#ifdef LIQUIDATE
	MarginCost = 0.15*Prices[0]/2;
#else
	MarginCost = comboStrike(2)/LEVERAGE/2;
#endif
```

LIQUIDATE uses another sort of **#define** for a 'compiler switch'. Such switches are a convenient way to select between different variants of the code. If the **#define LIQIDATE** is commented out, the code after **#else** is compiled. Bob then needs enough cash on his account for paying the assigned underlying position when his put or call gets exercised at the strike price. Since that's more than the options margin, Alice uses it for the margin cost. The **comboStrike** function returns the strike of the put, the second leg of the combination that got the higher strike and thus produces worse margin cost. **LEVERAGE** is Bob's leverage for buying or short selling stock. The amount is divided by 2 because Zorro assigns the margin cost to both options, but Bob needs only to fear that one of them is exercised. If both are, the long and short underlying positions will cancel out each other.

What happens if **LIQUIDATE** is not commented out? Alice assumed that Bob is greedy enough to risk liquidation for a better return. If an option is exercised, but Bob has not enough cash on his account, he'll get a margin call. The broker will then liquidate his assigned underlying position. That means selling it at market, and that's just what Bob wanted to do anyway. So the margin call and liquidation is actually a part of the strategy. Not all brokers like such a trading behavior, and Alice will advise Bob to discuss this with his broker to be sure. The **LIQUIDATE** switch then remains active, the code part after the **#ifdef** is compiled, and the margin cost is reduced to the original margin of the call/put combo. There's no cash reserve for covering the underlying position. That means less cost, more options, and more profit.

The rest of the script calculates the number of options and enters the combo:

```
var TotalCost = 2*Multiplier*MarginCost;
Lots = (Capital+ProfitClosed)/TotalCost;
if(Lots >= 1) {
   enterShort(Lots*comboLeg(1));
   enterShort(Lots*comboLeg(2));
}
```

Strategies like this one normally run without square root rule, since the investment amount is determined mostly by margin, not by drawdown. You also do not invest 1% or 2% of your capital at a time. You invest 100%. So the number of options is just determined by the available capital. The maximum cash requirement per combo is calculated in **TotalCost**, then the current **Capital** including backtest gains is divided by it for getting the number of contracts to sell and storing it in **Lots**. Since the investment amount is set up with the **Capital** slider, Bob can reserve some part of his account for

some other strategy. The **Lots** parameter, multiplied with the lots from the combination, is then directly passed to the **enterShort** calls.

The final line makes sure that the script terminates after it did its deed in live trading:

```
if(Live && !is(LOOKBACK)) quit("Ok!");
```

The result from the options system with 100% reinvesting and $10.000 initial capital:

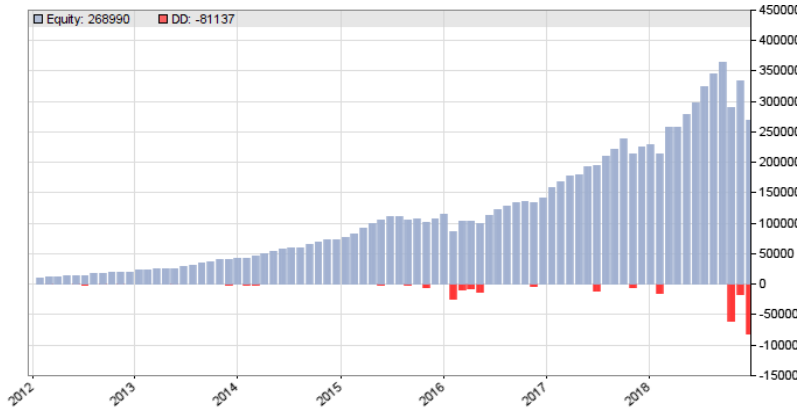


Fig. 42 – Selling options with reinvesting

The CAGR with this system is above 40%, despite the drawdown in fall 2018. Without the liquidation trick, it would be only in the 20% area.

6 tricks for more profit with options

► Selling options is better than buying options.

► Rather than using the current spot price as a center point of an option combination, take underlying trend into account.

► Determine the real margin cost of your options, at opening the position and at a possible exercising. It has a strong effect on your returns. For playing it safe, keep enough cash on your account for covering the underlying when they get exercised.

► If you're not the 'play it safe' type, use margin calls and liquidations to your advantage.

► Don't be afraid to sell options that are in the money. Due to the high premium, you can even make more profit with them. Verify this by backtest.

► Let short options expire. Early closing or 'rolling over' losing positions often reduces the overall profit. That's not investigated in this chapter, but it's the author's impression after backtesting many, many options strategies.

8 The strategy with no risk

Alice: "You're driving that Porsche again? Didn't you sell it?"
Bob: "I bought it back."
Alice: "So, trading went well recently?"
Bob: "You can say so. I also got my golden bathtub back."
Alice: "Then why that sullen face?"
Bob: "I have now some money – and that's the problem. What shall I do with it? I don't want to invest it back in my trading systems. Too much risk. I don't want to buy another Porsche. I can drive only one. I want to store the money in a safe haven, for hard times. But how? "
Alice: "Put it under your pillow."
Bob: "Inflation would eat it."
Alice: "Maybe a savings account?"
Bob: "Ridiculous."
Alice: "Bonds?"
Bob: "That's even worse. I'm a trader. So I want to my money in a trading system. A special one. With no risk. In the long term, the money must always grow, never shrink. Loss of money should be out of the question."
Alice: "That might be slightly difficult."
Bob: "That's why I'm paying your exorbitant fees. Make it possible."

The perfect stock portfolio

Which assets would a no-risk strategy trade? Obviously not Forex, because currencies undergo strong and unpredictable fluctuations. Options neither – there are option strategies with little risk, but they also achieve little profit. How about a portfolio of stocks? In the long run, stocks tend to rise. And it is unlikely that the value of major stocks suddenly drops to zero. But it must be avoided to keep a losing stock too long. Alice decided to manage a portfolio of stocks in a way that it always contains the stocks with the highest returns, and never contains stocks with negative returns. This appears to be the best way to fulfil Bob's requirements.

After some contemplating, she wrote this script (**Alice6**, agreed fee: 10% of Bob's profits):

```
#define RISK   0.7

var Momentums[100],Weights[100];

void rotate(int GoLong)
{
   var Invest = RISK*(Capital+ProfitTotal);
   for(listed_assets) {
      int NewLots = Invest*Weights[Itor]/MarginCost;
      if(Test) {
         if(NewLots > LotsPool && GoLong)
            enterLong(NewLots-LotsPool);
         else if(NewLots < LotsPool)
            exitLong(0,0,LotsPool-NewLots);
      } else if(Live) {
         int OldLots = brokerCommand(GET_POSITION,Asset);
         if(NewLots > OldLots && GoLong)
            enterLong(NewLots-OldLots);
         else if(NewLots < OldLots)
            enterShort(OldLots-NewLots);
      }
   }
}

void run()
{
   set(PLOTNOW|LOGFILE);
   StartDate = ifelse(Live,NOW,20120101);
   BarPeriod = 1440;
   LookBack = 100;
   SaveMode = 0;

   assetList("AssetsSP50");
   Capital = slider(1,10000,0,20000,"Capital","");

   for(listed_assets) {
      asset(Asset);
      vars Prices = series(priceClose());
      Momentums[Itor] = ROC(Prices,LookBack-1);
   }

   if((Live && !is(LOOKBACK))
```

```
      || (tdm(0) == 1 && month(0)%2))
   {
      distribute(Weights,Momentums,NumAssetsListed,5,0.5);
      rotate(0);
      rotate(1);
   }
   if(Live && !is(LOOKBACK)) quit("Ok!");
}
```

Like the options system from the previous chapter, this is again a script that is only started at certain dates – in this case at the first working day every second month. Let's look into some new stuff in the run function.

```
assetList("AssetsSP50");
```

That command loads an asset list that contains the 50 major US stocks and is set up for history download from Stooq. Stooq is a free online source for US and UK stocks and ETFs. So when that script is started, the first thing it will do is downloading all 50 asset histories.

```
for(listed_assets) {
   asset(Asset);
   vars Prices = series(priceClose());
   Momentums[Itor] = ROC(Prices,LookBack-1);
}
```

The **for(listed_assets)** statement is a **for** loop that we already know from the coding chapter. It runs through all assets in the **AssetsSP50** list that we just loaded, and sets the **Asset** string to the current asset name. The history downloading happens in fact in this loop by the **asset** call. A price series is creates. The **ROC** indicator then calculates the price change in percent since **(LookBack-1)** days ago. We must subtract 1 from the lookback period because **ROC** needs 1 more bar; too short lookback periods are indicated with an error message. The rate of change is then stored in the global array **Momentums** that Alice had defined at the begin of the script:

```
var Momentums[100],Weights[100];
```

An array is a list of objects, in this case a list of 100 **var**s. Large arrays in scripts are usually defined global, i.e. outside functions. This has technical reasons, and has also the advantage that other functions can access the arrays. In a normal program, fixed-length global arrays would be a bad programmer

faux pas, but in strategy scripts we can make it easier. The **Itor** in the above loop is the number of the addressed element in the **Momentums** array, and identical to the number of the asset in the asset list. So any asset has an assigned element in this array.

```
if((Live && !is(LOOKBACK))
   || (Test && tdm(0) == 1 && month(0)%2))
{
```

That's probably the most complex **if** condition in this book. If either in live trading mode and after the lookback period, or in the backtest and at the first trading day of any other month, do something. If you write conditions like these, be careful with the brackets. The **tdm** function returns the count of the trading day of the month, and the **%2** is the modulo operator that yields the remainder of a division by 2. So **month(0)%2** is 1 when the month is not divisible by 2, and 0 otherwise.

```
distribute(Weights,Momentums,NumAssetsListed,5,0.5);
```

That's the core of the strategy. The **Momentums** array now contains the rates of change of all stocks. The distribute function takes this array and looks for the 5 greatest positive elements. It copies their values over into the **Weights** array, normalized so that their sum is 1. All the other elements of the **Weights** array are set to 0. The parameter **5** is the maximum number of nonzero weights. The **0.5** is a weight cap – no stock gets more than 50% weight in the portfolio. If there are not enough positive momentums, the weights may sum up to less than 1.

```
rotate(0);
rotate(1);
```

The **rotate** function is then called twice, first with parameter 0, then with 1. It is responsible for the actual trading:

```
void rotate(int GoLong)
{
   var Invest = RISK*(Capital+ProfitTotal);
   for(listed_assets) {
      int NewLots = Invest*Weights[Itor]/MarginCost;
      if(Test) {
         if(NewLots > LotsPool && GoLong)
            enterLong(NewLots-LotsPool);
         else if(NewLots < LotsPool)
            exitLong(0,0,LotsPool-NewLots);
```

```
      } else if(Live) {
         int OldLots = brokerCommand(GET_POSITION,Asset);
         if(NewLots > OldLots && GoLong)
            enterLong(NewLots-OldLots);
         else if(NewLots < OldLots)
            enterShort(OldLots-NewLots);
      }
   }
}
```

This function converts the weights to a corresponding number of shares, then rebalances the portfolio by entering or exiting positions. The **RISK** definition determines the amount of capital to invest. Too much and Bob risks a margin call when prices go down. Alice has set it to **0.7**, so stocks are allowed to drop by 30%.

In live trading mode, Alice must also take into account that Bob (or the broker) might have manually closed positions inbetween for some reason, or opened new ones. So the script does not store trades, but reads open position sizes from the broker API with the **GET_POSITION** command. The positions are then rebalanced by either short or long trades until any position size is equal to the calculated **NewLots**. In backtest mode the open position size is automatically stored in the **LotsPool** variable, and balancing happens with entering or exiting positions.

If the **rotate** function is called with the **GoLong** parameter at 0, positions are only decreased, never increased. This allows Alice to call it first with 0 for closing old positions, then with 1 for entering new positions. If a position were entered before another one is closed, the available capital could be temporarily exceeded and cause rejection of trades.

The result of the stock rotation system with 2:1 leverage and linear reinvestment:

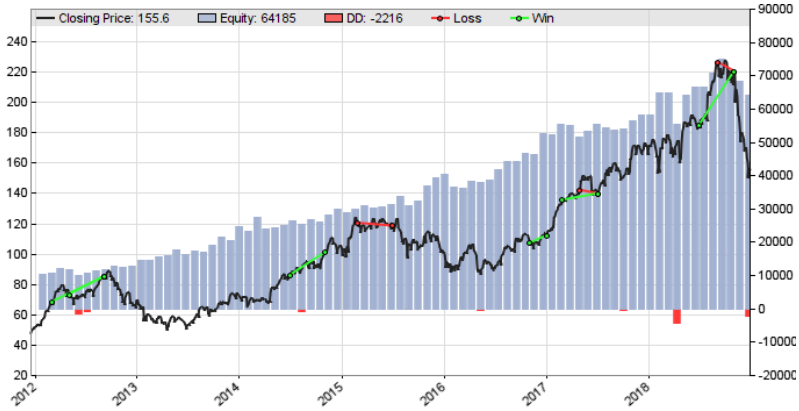


Fig. 43 – Stock portfolio rotation with reinvesting

This system produces about 30% CAGR. It's not really 'no risk' – when the economy crashes and all 50 stocks tank, the system will go out of the market, but must swallow any preceding loss. Still, 30% return is not bad for 5 minutes work every 2 months. Better for Bob than putting the money under the pillow.

5 tricks for stock trading systems

► Select and rank stocks simply by momentum. The concept that "stocks rising last year will drop this year" has no empirical foundation.

► Don't invest in stocks with recent negative momentum, even when they are cheap and even when this means going completely out of the market at certain times. Unless you know something about that stock that no other market participant knows.

► When your system has no open stock positions due to a market crash, consider parking your cash in bonds or treasuries. They probably won't produce much return, but any serious market crash smashes a couple brokers. Positions could be transferred to another broker. Any cash in your broker account would be gone.

► Don't invest all capital, unless you have a cash account. Unlike the options system from last chapter, stock trading systems on a margin account need a cash reserve for riding out drawdowns.

► Rebalance rarely, only once per month or any second month. Too frequent rotations will reduce the profit due to transaction costs and ask-bid spread.

9 Anatomy of a Scam Robot

Bob: "May I ask you a personal question?"
Alice: "What question?"
Bob: "Are you an over honest person?"
Alice: "When I recall our contracts, I charged about 30K for 300 lines of code. Does that answer your question? "
Bob: "Well, let's forget those contracts. The thing is, I need your help now to get rich."
Alice: "Aren't you already earning good money?"
Bob: "I don't want good money, I want to get rich. I mean really, obscenely rich. I want my own island in the Caribbean, and I feel that for this I'll need other methods. I thought about offering extremely expensive trading seminars, but I'm not a good speaker. I could write trading books, but I'm not a writer. So, my idea is to sell a trading robot. I already have a name: 'Forex Turbo Growth Pilot'. How does this sound?"
Alice: "You want me to program a scam robot? For ripping people off?"
Bob: "Ripping off... that's an ugly word. I mean, what can someone expect when he buys a robot for 99 dollars? If that thing really worked, it would be worth millions. I just want you to program a robot that looks better than all the other robots."
Alice: "And it does not matter if it really generates profit or not?"
Bob: "It's a robot after all. It must of course appear as if it would spit out money like mad, but it needs not really do that. Otherwise I would naturally use it myself and would not sell it."
Alice: "That makes programming it sort of easier."
Bob: "I have already ordered the advertising: 95% win rate! 25,000 pips annual profit! Confirmed with real trading! Your robot must produce those figures. The equity curve must go straight up to the sky, and it must come from a year verified live trading. A backtest won't do. Sadly, today people are mistrustful. Many won't buy a robot that has no excellent live trading history in myfxbook. "
Alice: "So I have to program a robot that is not profitable, but nevertheless has 95% win rate and produces 25,000 pips per year on a real money account?"
Bob: "You got it. Can you do that?"
Alice: "Sure. Piece of cake. "

Bob: "But I guess you'll still charge an outrageous fee?"
Alice: "I have to. This job will give me a guilty conscience. This must be financially compensated, at least."

The robot script

For unknown reasons, traders are more susceptible to fall for scam than other groups of the population. Trading robots are scripts, programs, or "Expert Advisors" with fancy names offered on many more or less obscure websites. They promise the buyer great wealth through algorithmic trading. A robot script works in a different way than a normal trading strategy. The least important part of the script is the algorithm for the trade signals. Robots usually use some conglomerate of classic indicators for this. Of course their developers are aware that solid profits by such systems are rather unlikely. But that does not really matter, for reasons that will soon become clear.

Alice has chosen a rather plain trade algorithm. This is her first robot script (agreed fee: $20,000):

```
function run()
{
  if(random() > 0)
    enterLong();
  else
    enterShort();
}
```

This laconic strategy enters a random trade on any bar. The **random** function returns in 50% of all cases a number greater than 0, therefore triggering a long trade; in the other 50% it will trigger a short trade. If trading had no cost, this strategy had a profit expectancy of zero. Selecting EUR/USD and clicking **[Test]** however reveals an average loss of about 2 pips per trade. This loss results from the simulated broker's ask-bid spread plus commission. So, no surprise here.

Such a random trading robot obviously won't sell well. Alice must pimp it up. The first step is setting up some system parameters for fulfilling Bob's 95% win rate demand (**Alice7.c**):

```
function run()
{
```

```
  BarPeriod = 1440;
  LookBack = 0;

  Stop = 200*PIP;
  TakeProfit = 10*PIP;

  if(NumOpenTotal == 0) {
    if(random() > 0)
      enterLong();
    else
      enterShort();
  }
}
```

The robot shall trade once per day, so Alice needs a 1440 minutes bar period. Backtest is restricted to simulate the year 2016 - the robot must work for one year only, so a longer backtest is not necessary. It also uses no lookback period, as there's no indicator or other function that would need any price history. This makes the parameter setup sort of easy.

The next lines are sufficient for the 95% accuracy:

```
Stop = 200*PIP;
Takeprofit = 10*PIP;
```

This establishes a risk/reward ratio of 20, through a stop loss at 200 pips distance from the current price, and a profit target at 10 pips distance. Each trade therefore risks 200 pips loss for a potential 10 pips reward. But this also means that the profit target will be hit 20 times earlier than the stop loss, and thus 20 times more often. From 20 random trades, about 19 will be won and only one will be lost - this is the 95% accuracy that the robot needs for matching Bob's advertisement.

For enforcing the win rate in this way, Alice must not use too small stop and profit target distances. **Stop** has higher priority than **TakeProfit** in the backtest. If both are triggered inside the same bar, **Stop** wins and the overall result becomes too pessimistic. This effect could be reduced by tick-precise simulation (**TICKS** flag), but at the disadvantage of a slower backtest. Therefore, Alice has set a relatively distant **Stop** at 200 pips for not triggering both limits too often simultaneously.

For achieving 95% accuracy, any trade must end with hitting either the stop loss or the profit target. Other exits are not allowed, since they would spoil

the 95%. Since a reversal - entering a trade in reverse direction – would also automatically close the current trade in Zorro's default no-hedge mode, Alice prevents trade reversal by only entering a new trade when no trade is open:

```
if(NumOpenTotal == 0) {
  if (random()> 0)
    enterLong();
  sonst
    enterShort();
}
```

The predefined variable **NumOpenTotal** is the current number of open trades.

A click on **[Test]** reveals that the current script version has indeed about 95% win rate. But of course this does not put it in the black. 19 out of 20 trades are won, but the loss from the 20th eats all the profits from the 19 winners before:

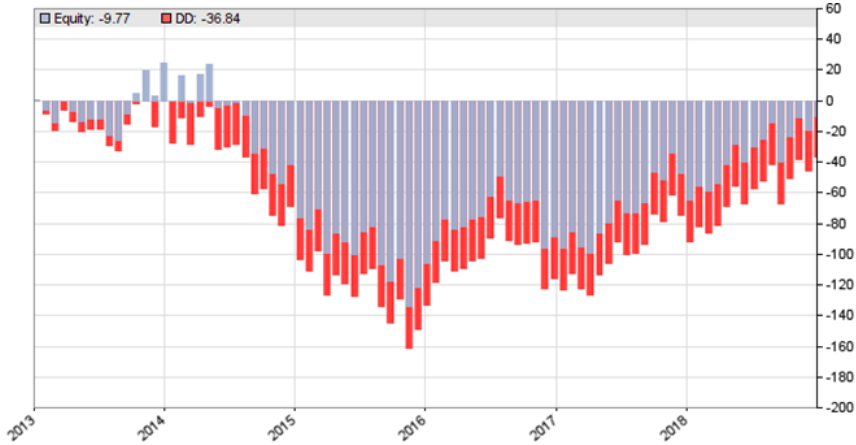


Fig. 44 – High win rate, lousy result

We can see that the win rate tells nothing about the performance of a system. Can such a strategy with random trades actually win in some situations? Yes, it can, when cartain inefficiencies appear in the price curve. If the price tends to reverse to the mean and strong movements are rare, this system can actually work better than totally random trading. The EUR/USD has indeed a mean reversion tendency in some years and on some time frames, but not enough to compensate transaction costs. Alice's system is still a loser.

But Bob wants an annual return of about 25,000 pips - enough to arouse expectation of great wealth, and to sell lots of robots. How can Alice adjust the script to generate this profit - real money, from live trading - with an obviously unprofitable strategy?

For this she has to use the very black tricks - the tricks of statistics.

How to cheat with statistics

The average loss of a random trade is spread and commission. Thus, one trade per day and 2 pips trading costs will produce about 500 pips average loss per year. This does not mean that every random trader will have 500 pips less on his account by the end of the year. Some might end up with a much larger loss, others even with a profit. Let's give 3000 traders some initial capital and the task to enter random trades, one trade per day, for many years. At the end of that time we'll put together their profits or losses in a histogram. It will look like this:

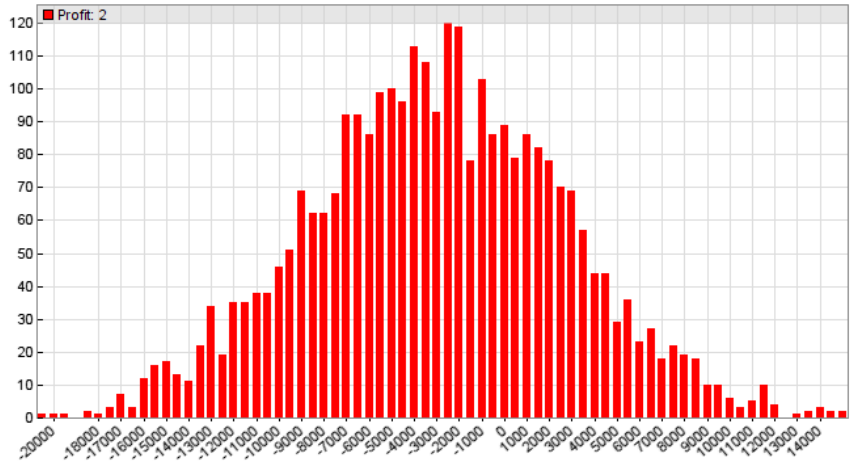


Fig. 45 – Profit distribution with random trading

This is again a Gaussian Normal Distribution - the famous "Bell Curve" that we already encountered in one of the previous chapters. It's from 3000 six-year simulation cycles of Alice's first random trading strategy (without the high win rate). The x axis of the chart is the profit or loss in pips at the end of the year. The y axis shows the number of traders that got that particular profit or loss. We can see in the above image that the largest group of traders in the middle had a loss of about 3000 pips, as expected – six years at 500 pip

each. But there are also some especially unfortunate traders with more than 20000 pips loss at the left side of the diagram. But on the other hand, far on the right side of the chart, a lucky few made more than 15000 pips profit! No doubt, those guys will consider themselves genius traders and brag with their success on trader forums...

You can see that the resulting chart above also debunks a widespread myth in the trader scene. It is a "known fact" that 95% of all private traders lose all their money in the first 12 months. Not true - at least not with completely random trading and one trade per day. You can see from the profit distribution that only about 55% lose money at all (the sum of the red bars with negative profit), while 45% end their trading period with a profit. Of course, they won't attribute their success to the bell curve, but to their trading skills.

This bell curve does not reflect the reality. Traders do not throw a coin. They normally follow either some system they've learned from books, or their gut feeling. This does not change the win chance, but it changes the form of the curve. The profit distribution of real traders is not a Gaussian, but a Lévy distribution. It has a smaller peak and fatter tails. That means the losers lose more, and the winners take more than in a random-trading situation. The winners are more likely to be found among employed traders, the losers among the private traders. Since they trade irrationally, take profits too early, and hold losses too long, the inefficiencies and anomalies in the price curve are often not to their advantage, but even to their disadvantage. In addition, their trades usually last only a few hours, so the trade costs get more weight. As a result, private traders have a slightly higher annual loss rate of about 65%, but certainly not 95%, not even beginners.

Let's see what happens when Alice uses her stop and profit targets for getting the 95% win rate. For plotting a profit distribution, she has modified her script a bit:

```
function run()
{
...
   NumTotalCycles = 3000;
...
}

function evaluate()
{
   plotHistogram("Profit",ProfitClosed/PIPCost,100,1,RED);
   static var AllProfits; static int AllTrades;
   if(TotalCycle == 1) { AllProfits = 0; AllTrades = 0; }
```

```
   AllProfits += ProfitClosed/PIPCost;
   AllTrades += NumWinTotal + NumLossTotal;
   if(TotalCycle == NumTotalCycles)
      printf("\nAverage Profit: %0.1fp  Per Trade: %0.1fp",
         AllProfits/NumTotalCycles,AllProfits/AllTrades);
}
```

For producing the histogram, active the **HISTOGRAM** compiler switch in the script. Like the reality check script, this one again generates a histogram of the profits, this time with 3000 simulation cycles for a smoother curve. Alice has divided the **ProfitClosed** result by the predefined variable **PIP-Cost**, the profit or loss in $ when the price of the asset changes by 1 pip. This converts profit to Pips. Each bar in the histogram now represents a profit range of 100 pips. The profit distribution with 95% win rate now looks quite different:

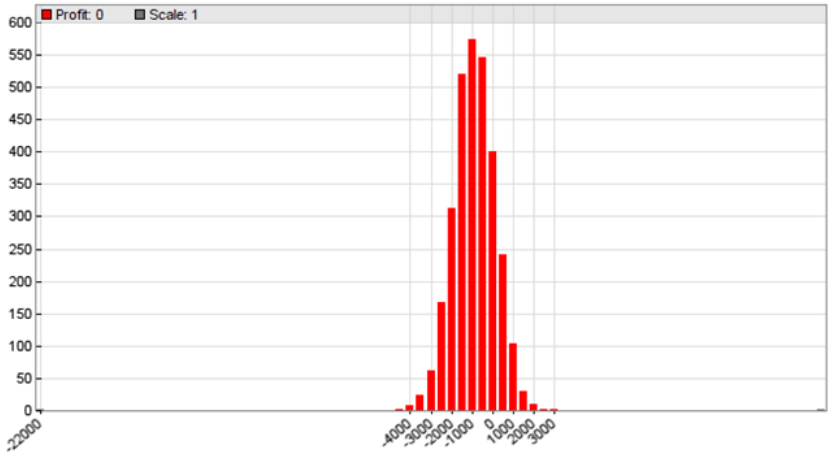


Fig. 46 – Profit distribution at 95% accuracy

We still have a kind of bell curve, which however is now strongly compressed in comparison with the previous one. Stop loss and profit target now prevent large profits and huge losses. With a profit target of 10 pips, no trader can take more than 2500 pips per year, even in the unlikely event that all trades are won. Alice, however, needs at least a 10-fold higher annual result. She can do nothing against the average loss of 500 pips. But she can manipulate the profit distribution in such a way that as many traders as possible reach 25,000 pips at the end of the year. This is possible with a special trade method, which robots like to use.

The martingale method

Alice has added three more lines to her strategy (activate the **MARTINGALE** compiler switch):

```
var ProfitGoal = 100*PIP*Bar;
Lots = (ProfitGoal-ProfitClosed*PIP/PIPCost)/TakeProfit;
Lots = clamp(Lots,1,100);
```

That's a martingale system. Such systems are often used by trading robots, signal providers, and beginners in roulette: you bet on red or black and double your stake after every loss. Alternatively, you open two new positions for every lost trade. The theory is that a loss increases your win chance the next time. Unfortunately, it won't. On the contrary, market inefficiencies are often autocorrelated, so after losing a trade there's a good chance that you'll also lose the next one. Although martingale systems at first indeed make steady profits, even when trades are entered at random. After some time, maybe a couple months, they will encounter a long loss streak and wipe out the account[1]. As we remember, drawdowns and losses grow with the square root of time and will therefore some day – very soon when the investment is increased overproportionally, as in a martingale system - exceed the available capital.

This does not worry Alice much. After all, it's not she who has to trade with that system. At first Alice determines a profit goal. She needs 100 pips per day for ending the year with more than 25,000 pips profit (there are 252 trading days in a year). A day is equivalent to a bar, so at any bar the accumulated profit should be 100 pips times the bar number. So at the first day the robot should have accumulated 100 pips, at the second day 200 pips and so on. The expected profit is stored in the **ProfitGoal** variable.

The next line is the martingale. The lot size is set dependent on how much the current profit in pips (**ProfitClosed * PIP/PIPCost**) deviates from the profit goal. The factor **PIP/PIPCost** converts profit in account currency

[1] The average lifetime of a martingale system is calculated in the next chapter.

units into a price difference. If the profit is negative or far below the goal, Alice needs a huge lot size to catch up. The number of **Lots** is calculated so that the next winning trade reaches the profit goal. The profit per lot of a winning trade is **TakeProfit** (disregarding spread and commission). So the number of lots is the difference to our profit goal divided by **TakeProfit**. Since this means a heavy increase of the trade volume after any lost trade, it is a martingale system.

The **clamp** function limits **Lots** between 1 and 100. We need at least 1 lot per trade, and we don't want to exceed 100 lots for not using crazy trade sizes. When analyzing robot strategies, one can notice such a martingale system from telltale peaks in the lot size. For this reason, robots or signal providers often increase not the number of lots, but the number of trades, which is less suspicious.

Aside from the martingale code, Alice has now restricted the backtest to 1 year, because that's the time period for Bob's real account trading. The smaller the time, the higher the chance to survive a martingale. Alice also selected the year 2015 for the test because that was a year with average return of that strategy. Other years produced better or worse results.

Every click on **[Test]** will now generate a different equity curve. Many look like this:

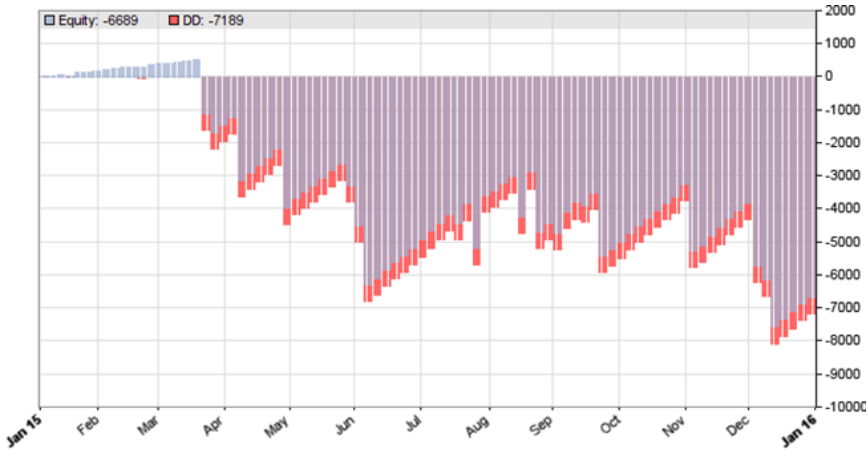


Fig. 47 – Martingale went south

But other curves – surprisingly many - look like this:

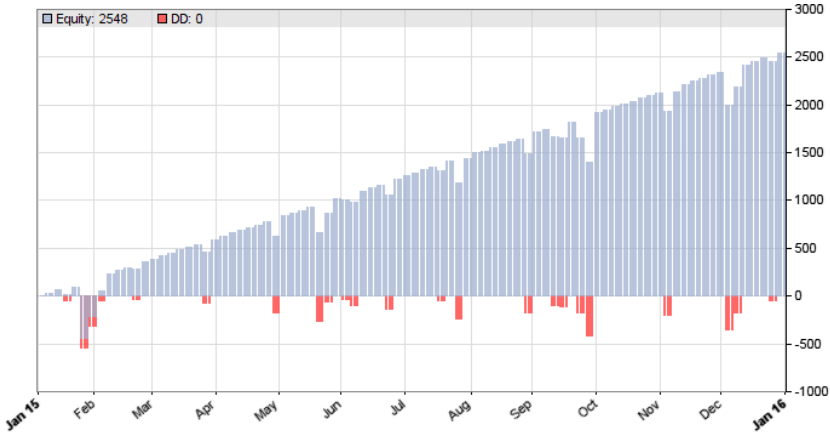


Fig. 48 – Martingale, lucky run

This is the perfect 1-year equity curve that Bob wanted for his robot, with 25,000 pips end result. The curve even a little too perfect - its straight slope comes from the **ProfitGoal** variable that just linearly increases with the bar number. For really selling the robot, Alice would have to modify the profit goal formula for letting the curve appear bumpier and more realistic. We leave that as an exercise to the reader…

Let's now look at the profit distribution with this modification after 1 year trading:

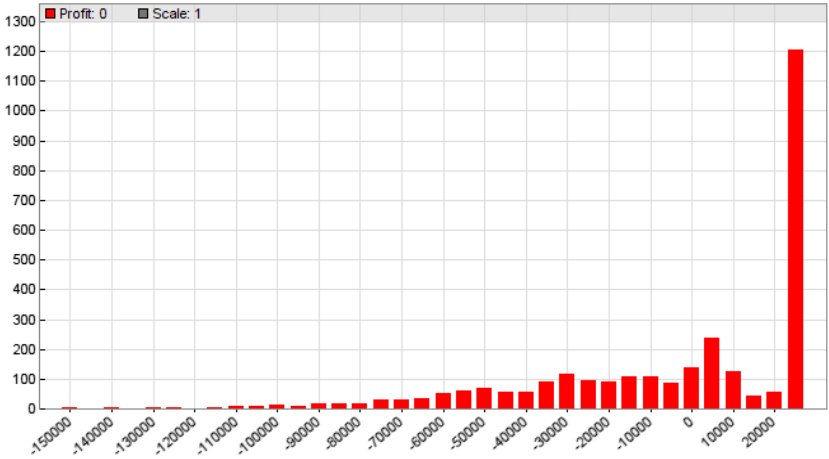


Fig. 49 – Martingale profit distribution

This distribution does not resemble a bell curve anymore. The distribution got an extremely long left tail and a sharp peak at the right in the 25,000 pips profit area. From our 3000 traders, 1200 earned 25,000 pips with this robot! Sadly, most other traders suffered painful losses, many even up to 150,000 pips. But we hope a merciful margin call saved them early.

A propos margin call: The profit distribution chart is in fact a bit misleading. The year won't end with 1200 lucky traders. Many of them will bite the dust before when their capital ran out. We'll simulate this (activate the **MARGIN-CHECK** compiler switch):

```
if(Equity < -250) {
   quit("Game over!");
   return;
}
```

We have equipped any of the 3000 traders with capital to ride out a $250 drawdown, but not more. Otherwise the trader is out of the game, and the **quit** call prevents further trading until the end of the simulation. This changes the profit distribution noticeably:

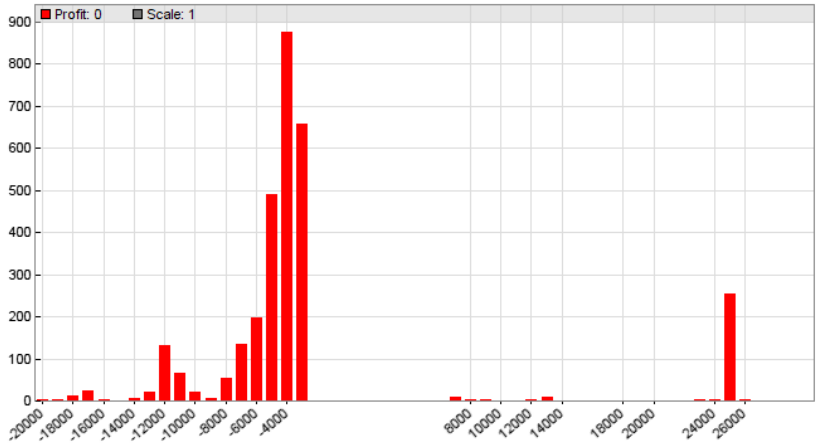


Fig. 50 – Profit distribution with margin call

Of the 3000 traders, about 250 still reached the 25,000 pips goal without a fatal drawdown inbetween. And this with totally random trading!

Alice has now a script that indeed generates more than 25,000 pips per year. There's a slight problem though: it works only for one out of 12 traders (250 of the 3000). Many of the rest will also earn large profits in the first months

due to the martingale system and the high win rate, but they all will have been hit by a margin call before the end of the year. Often they even have to re-margin their broker, since their fatal last trade ended with a much bigger loss than $250.

Bob won't mention this little problem in his robot advertisement - but he'll need something else. For selling the robot, at least one of those 250 profitable equity curves has to be verified on a real account by a trusted trade recording service. For this purpose Bob will invest $10,000. Not, as you might think, for bribing the service to confirm a fake equity curve. No, the $10,000 are used for real trading.

Wealth from a losing system

Bob's next steps to insane wealth:

► Get the script from Alice. Pay her fee.
► Open 20 trading accounts, each with $250 deposit.
► Register the accounts with a trade recording service such as myfxbook™.
► Start his scam robot script on all accounts.

For financing the 20 accounts, Bob invests half of his $10,000. The money is not lost. A part of it can be recovered later. But first Bob needs strong nerves, as $250 deposit leave not much room for drawdowns. Most of the 20 accounts will sooner or later go down with a margin call. No problem for Bob: for any wiped account, he just opens a new one with $250, until the rest of the $10,000 is spent.

Now Bob has to wait a year.

$10,000 allow running about 40 accounts simultaneously. If the equity curve of an account does not look good, even if the account is in profit, Bob just closes it and opens a new one. After a year, Bob has only 3 accounts left, but any one with about 25,000 pips profit, equivalent to about $250 at the selected leverage. Because the accounts had been started with only $250 deposit, myfxbook has given them all impressive annual gain rates of more than 1000%.

Bob keeps the account with the smoothest equity curve and closes the rest. A part of his $10,000 is thus recovered and goes back on Bob's bank account.

Now Bob has the verified equity curve that he wanted, and can start advertising and selling his robot. And the money will come rolling in, and he will get obscenely rich. And he will retreat to his Caribbean island and live from his robot sales happily forever after. And when his robot won't sell anymore because too many people lost their money with it, he'll have already 10 new robots in preparation.

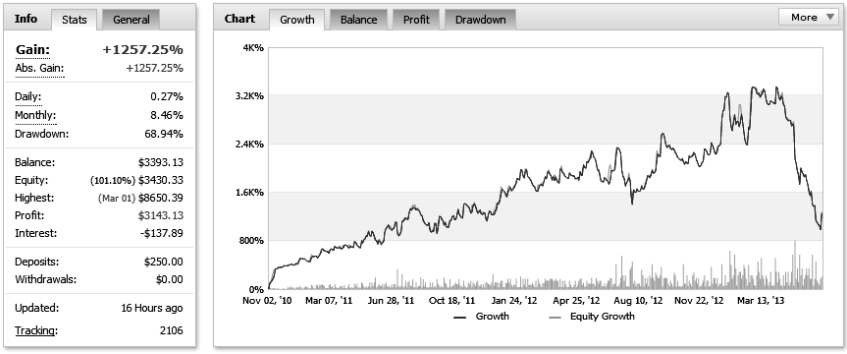


Fig. 51 – Equity curve of a real robot (screenshot: myfxbook)

The above equity curve was generated by a typical commercial robot that was very popular in the trading scene, sold like hot cakes, and got enthusiastic reviews due to its "myfxbook-verified trading record". We won't tell the name of this robot, and we're certainly not suggesting that something was fishy with its equity curve. Only a little curious is the fact that the account was started with only $250 deposit – a bit small for seriously testing a system. Or that the lot size - the light grey bars - grew large peaks whenever the equity went down, just like a martingale system. Or that the equity curve went straight upwards in its first part, but suddenly - just after the robot started selling - dropped like a stone, losing all previous 'profits'. Really strange. Shame upon him who thinks evil upon it!

Conclusion

The purpose of the chapter was not to learn ripping off other traders. It just should make you aware of a typical phenomenon in the trading scene. Are all commercial robots 100% scam? This is hard to tell. A honest robot is theoretically possible, but it would not sell well: the expectations of robot buyers require high win rates and straight equity curves. This can not be achieved

with a real trading system. Therefore, vendors have no interest in selling robots that really work, even if they knew how to program them. They get much better reviews and more sales with scam.

If you know how such a robot works, you'll be able to see the hints in the performance figures. Few robots are programmed so good that they can not be identified as scam at a first glance on their trading curves. Aside from that, programming a scam robot is a good exercise in statistics, profit distributions, and risk and money management. So the knowledge of scam robot scripting helps with programming real strategies. Which will be the topic of the final chapter.

10 Developing own systems

The last chapters introduced various techniques and methods of algorithmic trading. All example strategies are simple and were coded with only a few lines of script. Zorro provides far more functions to detect and exploit all sorts of market inefficiencies. In particular, you can connect Zorro to the analysis software "R" and this way get access to the latest libraries of machine learning and artificial intelligence, such as support vector machines or "deep learning" neural nets. This however would be material for another book. You're invited to experiment with all features. I do the same on my blog http://financial-hacker.com – just drop in from time to time.

The first step to your own strategy is to train your ability to put an algorithm into code. After reading this book, you know the steps of strategy development and can read and understand code. But writing code is a different matter. Practice makes perfect. Some trader forums have sections where traders describe their ingenious systems. Try coding them. With some experience you'll soon be able to convert a simple system into code within 10 minutes, and test it. You probably may not encounter this way a great many of useful systems[1]. But you'll learn how to write them.

Trading signals often have complex setups and conditions. Don't program everything at once. Do it step by step, starting with the most basic trade entry condition, which you then, step by step, extend until the final strategy is complete. Test the strategy after every single step, look in the log, and check the trades. If they do not behave how they should, examine the behavior within the respective bars. Conventional "debugging" line by line is not very helpful with trading systems: better debug bar by bar. You can use Zorro's stepwise

[1] Don't attempt to explain to the trader why his system is no good. Confrontations with reality are not welcome all the time.

debugger for this. Watch the behavior of the relevant variables for every bar. Also, plot instructions for displaying the signals on the chart are useful.

The development process

Once you're proficient in coding, how can you proceed to develop a profitable trading system? Here's the skeleton of an ideal strategy development process, based on an article series from the Financial Hacker blog.

Find a price curve anomaly. Decide for a market inefficiency to exploit – or discover a new one. The best known inefficiencies are listed in the next chapter. Think about which price curve anomaly this effect could produce (an anomaly is any systematic deviation from randomness). Describe it with a quantitative formula or at least a qualitative criteria. You'll need that for the next step.

Do research. Find out if the hypothetical anomaly really appears in the price curves of the assets that you want to trade. For this you first need enough historical data of the traded assets – D1, M1, or tick data, dependent on the time frame of the anomaly. How far back? As far as possible, since you want to find out the lifetime of your anomaly and the market conditions under which it appears. Write a script to detect and display the anomaly in price data. If you find no clear signs of the anomaly, or no significant difference to random data, improve your detection method. And if you then still don't succeed, go back to step 1.

Do the basic algorithm. Write code that generates the trade signals for buying in the direction of the anomaly. A market inefficiency has normally only a very weak effect on the price curve. So your algorithm must be really good in distinguishing it from random noise. At the same time it should be as simple as possible, and rely on as few free parameters as possible.

Once you got the code, it's time for a first backtest. The precise performance does not matter much at this point – just determine whether the algorithm has an edge or not. Can it produce a series of profitable trades at least in certain market periods or situations? If not, improve the algorithm or write a another one that exploits the same anomaly with a different method. But do not yet use any stops, trailing, or other bells and whistles. They would only distort the result and give you the illusion of profit where none is there. Your algorithm must be able to produce positive returns either with pure reversal, or at least with a timed exit.

In this step you must also decide about the backtest data. You normally need M1 or tick data for a realistic test. Daily data won't do. The data amount depends on the lifetime (determined in step 2) and the nature of the price anomaly. Naturally, the longer the period, the better the test – but more is not always better. Normally it makes no sense to go further back than 10 years, at least not when your system exploits some real market behavior. Markets change extremely in a decade. Outdated historical price data can produce very misleading results. Most systems that had an edge 15 years ago will fail miserably on today's markets. But they can deceive you with a seemingly profitable backtest.

Implement a filter. No market inefficiency exits all the time. Any market goes through periods of different or totally random behavior. It is essential for any system to have a filter mechanism that detects if the inefficiency is present or not. The filter is at least as important as the trade signal, if not more – but it's often forgotten in trade systems. You can see in chapter 3 how filtering by the Market Meanness Index improved the system.

But this also bears a danger: Don't add a filter just because it improves the test result. Any filter must have a rational reason in the market behavior or in the used signal algorithm. If your algorithm only works by adding irrational filters: back to step 3.

Don't add too many or too complicated filters. An incomprehensible mess of conditions and filters with numerous parameters is not only a source of errors. It also makes optimizing in the next step difficult. If additional conditions do not lead to considerable performance improvements, drop them. But if they improve the performance too much, ask why. Maybe you've found a new inefficiency. Or you got trapped by an error. Always expect the latter.

Optimize gently. Optimizing strategy parameters is a big opportunity to fail without even noticing it. In fact, a recent study (Wiecki et.al. 2016) showed that the better you optimize your parameters, the worse your system will fare in live trading! The reason of this paradoxical effect is that optimizing to maximum profit fits your system mostly to the noise in the historical price curve, since noise affects result peaks much more than market inefficiencies.

Not optimizing at all however is no solution, since you then won't know the dependence of your system to parameter changes. So, optimize mildly, and don't select sharp profit peaks, but gentle hills for your parameters.

Do out-of-sample analysis. Of course the parameter optimization improved the backtest performance of the strategy, since the system was now

better adapted to the price curve. So the test result so far is worthless. For getting an idea of the real performance, we first need to split the data into in-sample and out-of-sample periods. The in-sample periods are used for training, the out-of-sample periods for testing. The best method for this is Walk Forward Analysis with a rolling window into the historical data. For preventing that it still produces too optimistic results just by a lucky selection of test and training periods, it makes sense to perform WFA several times with slightly different starting points of the simulation. If the system has an edge, the results should be not too different. If they vary wildly: back to step 3.

Do a reality check. Even though the test is now out-of-sample, the mere development process – selecting algorithms, assets, test periods and other ingredients by their performance – has added a lot of selection bias to the results. Are they caused by a real edge of the system, or just by biased development? Determining this with some certainty is the hardest part of strategy development. There are several methods at your disposal.

1. **White's Reality Check.** That's the most reliable method to find out how your system would perform in live trading (look for details on the Financial Hacker blog). But it's also the least practical because it requires strong discipline in parameter and algorithm selection. Other methods are not as good, but easier to apply.

2. **Montecarlo.** Randomize the price curve by shuffling without replacement, then train and test again. Repeat this many times. Plot a distribution of the results (an example of this method can be found in chapter 6). Randomizing removes all price anomalies, so you hope for significantly worse performance. But if the result from the real price curve lies not far east of the random distribution peak, it is probably also caused by randomness. That would mean: back to step 3.

3. **Variants.** It's the opposite of the Montecarlo method: Apply the trained system on variants of the price curve and hope for positive results. Variants that maintain most anomalies are oversampling, detrending, or inverting the price curve. If the system stays profitable with those variants, but not with randomized prices, you might really have found a solid system.

4. **Really-out-of-sample Test.** While developing the system, ignore the last year (2019) completely. Even delete all 2019 price history from your PC. Only when the system is completely finished, download the data and run a 2019 test. Since the 2019 data can be only used once this way and

is then tainted, you can not modify the system anymore if it fails this test. Just abandon it. Assemble all your metal strength and go back to step 1.

Implement risk management. Your system has so far survived all tests. Now you can concentrate on reducing its risk and improving its performance. Do not touch anymore the entry algorithm and its parameters. You're now optimizing the exit. Instead of the simple timed and reversal exits that we've used during the development phase, we can now apply various trailing stop mechanisms. For instance:

Instead of exiting after a certain time, raise the stop loss by a certain amount per hour. This has the same effect, but will close unprofitable trades sooner and profitable trades later. When a trade has won a certain amount, place the stop loss at a distance above the break even point. Even when locking a profit percentage does not improve the total performance, it's good for your health: observing profitable trades wander back into the losing zone can cause serious ulcers.

Of course you now have to optimize and run a walk forward analysis again with the exit parameters. If the performance didn't improve, think about better exit methods.

Implement money management. Money management algorithms reinvest your profits and/or distribute your capital among portfolio components. Dependent on whether you trade a single asset and algorithm or a portfolio of both, you can calculate the optimal investment with several methods. There's the OptimalF formula by Ralph Vince, the Kelly formula by Ed Thorp, or mean/variance optimization by Harry Markowitz. In chapter 6 we've learned different reinvestment methods, but usually you won't hard code them, but calculate the investment volume externally. This allows you to withdraw or deposit money without the need to change your code.

Important: Do not reinvest the profits during the test runs. Do that only at the end when your strategy is complete. Until then, test with fixed lot sizes onyl. Reinvesting can distort the equity curve completely and can affect the result to a point that it useless.

Prepare for live trading. You can now create the user interface of your trading system. Determine which parameters you want to change in real time, and which ones only at start of the system. Provide a method to control the trade volume, and a 'Panic Button' for locking profit or cashing out in case of bad news. Display all trading relevant parameters in real time, and make sure that you can supervise the system from wherever you are, for instance through an online status page. If you want to run it for a long time, add buttons for re-

training and provide a method for comparing live results with backtest results, such as the Cold Blood Index described on the Financial Hacker blog.

Strategies that work

Despite or perhaps precisely because of algorithmic trading systems, inefficiencies in the financial markets have increased in the last decade. However, they are more difficult to recognize than they used to be, as more influences affect the prices. With the right methods, however, algorithmic trading is now more lucrative than ever before. Here you will find a list of inefficiencies that are all more or less suited to produce a regular income:

Trend following. The classic. For following the trend you need to know it beforehand, so those strategies usually try to predict a future price trend from the current one. But often it is enough to simply keep up with the current trend of a price curve. Traders tend to follow the crowd: Once some buy, others start buying too. This gives the trend a certain self-sustaining element and a certain inertia. Once it has started, it goes on for a while. Trend following is one of the most frequently used strategies, but also one of the most problematic. The crucial point is to recognize a long-term trend as early as possible and to distinguish it from short-term price movements - two contradictory requirements that must be reconciled. If this does not work, the system generates losses instead of profits. Algorithmic trend-following systems tend to exacerbate existing trends and in this way destabilize the market, particularly in the case of very fast reacting systems.

Counter-Trend (Mean Reversion). Traders often believe in a "fair price" of an asset, especially if it was predicted by a stock exchange guru in a TV show. They buy when the actual price in their opinion is too low, and sell when the asset appears too expensive. This pulls the price back to a mean value after running for a time in a certain direction. The mean value itself can often be subject to a trend, which is however relatively easy to filter out. Mean-reversion systems are among the most reliable work horses of algorithmic trading. They have the additional advantage of counteracting short-term trends and stabilizing the markets.

Cycles. With spectral analysis, the dominant cycle - or even several cycles - can be filtered out of a price curve; an example of this method was already the topic of a previous chapter. The cycle lengths of some assets often remain relatively constant over some time, and they can regularly produce profits during this period. Important is a filter algorithm that detects at an early stage when cycles from the price curve disappear and new, different cycles emerge.

Clusters. Traders often envision imaginary lines on the price chart that stop a drop of the price ('support') or pull it down when it rises ('resistance'). The alternative terms 'supply' and 'demand' for the respective lines have recently come into fashion. Selling at the resistance and buying at the support line causes a classic self-fulfilling prophecy. It generates price clusters, i.e. the concentration of prices at certain levels, visible as peaks in the price distribution. An algorithmic system does the same thing the traders do - but a bit faster and more accurately.

Curve patterns. In the belief of many traders, changes in the price curve are preceded by certain curve patterns. Most of them, such as the famous 'head-and-shoulders' pattern that supposedly indicates a trend change, are myths and can not be statistically found in price curves. But some simple patterns, for example 'cups' or 'half-cups', i.e. cup-shaped bulges of the curve just before a steep trend movement, do really exist. They can be attributed to expectations of traders. Such patterns can be exploited for trade signals by detection methods such as the Fréchet algorithm.

Price Action. Certain combinations of the ranges and changes, or of the open, close, high and low prices of 3 or 4 consecutive candles sometimes precede a short-term price movement. In contrast to the traditional 'Japanese candle patterns', this effect can actually be demonstrated in a number of assets. The inefficiency occurs with daily and hourly candles. The patterns are not fixed – they are different for each asset and have only a lifetime of a few years. Their cause is not clear; perhaps it's certain regularities in the behavior of less large market participants. In any case the effect is rather weak, so despite the evidence that those short-lived patterns exist, and despite the many software tools to exploit them, really successful systems with this method have not yet emerged.

Price limits. Sometimes a currency is wholly or partly linked to another to achieve certain economic objectives. A well-known example was the above mentioned EUR/CHF price cap, which was established by the Swiss National Bank and supported by massive sales. Knowing that an asset can not exceed a certain price level is extremely valuable for a trading system. It allows for lucrative trade methods, which would otherwise not be advisable because of the high risk, such as grid trading. Any artificial price limit can be exploited in trading systems.

Price shocks. They often come Monday or Friday morning when companies or organizations publish good or bad news, causing prices to suddenly rise, or to drop like a stone. Examples were the Brexit or the removeal of the

Swiss-Franc cap. But even without knowing the cause of the price jump, a system can recognize the first signs and, if necessary, jump quickly onto the bandwagon. Or it can wait for the peak value of the price jump and then enter at the first sign of a return swing, often resulting in high profits in a short time.

Seasonality. Price curves of stock indices and commodities are often subject to periodic fluctuations, not necessarily at the rhythm of one year, but also in monthly, weekly or daily rhythm. These fluctuations are caused by recurring changes in demand or supply. They can be used in strategies. US stock indices are often moving upwards in the first days of a month, and sometimes also show an upward trend in the early morning hours before the main trading hours of the day. Seasonality may even affect currencies: In May, the dollar is often rising as US stock traders are selling shares, realizing profits and generating dollar demand according to the *Sell in May and go away* Rule. Such seasonal effects can be detected by computer analysis in price curves.

Gaps. When traders have a night or a whole weekend to think about their next moves, their decisions are often predictable. A system can take advantage of this and predict the price of Monday morning from the price curve of Friday afternoon. This can work with currencies, equities, and equity indices, especially when they are traded only during certain market hours.

Statistical arbitrage. If two assets are strongly correlated or anticorrelated - if one rises, the other rises too, or vice versa - an artificial price can be derived from the difference between the two price curves. This price difference has has greater inefficiencies than the two original price curves and, in particular, often shows strong mean reversion. As long as the assets stay close to each other, their difference can be traded using similar lucrative methods as a price curve with an upper or lower price limit.

Smart Money. Follow the forecasts of informed market participants. This does not mean trading 'experts' or gurus, but big producers or buyers of a certain good or commodity. As an insider, you can often predict the price trend within certain limits, and hedge risk by purchasing futures or options of the respective asset. Such market activities are published by organizations - the most prominent publication is the Commitment Of Traders (COT) report, which is updated every few days on the Commodity Futures Trading Commission website and can be downloaded for evaluation. There is, however, no guarantee for the smartness of the relevant market participants.

Strategies that don't work

There are many ways to develop profitable strategies. But a lot more popular trade methods are known for letting the trader and his money go separate ways in surprisingly short time. Some of these methods appear to be nonse at first glance, but others are not so obvious. If a new trade method is introduced to you in a book, magazine or trader forum, you should always ask yourself (or the inventor of the method) two simple questions: What market inefficiency is it based on? And how was it tested? If there are no reasonable answers, stay away. However, virtually every abstruse method finds its followers who swear that it promises or even has alredy produced big profits. Which is not always a lie: there are also lottery winners, after all. This effect is called *Fooled by Randomness*. Here is an incomplete list of irrational trade methods:

Staring at price curves. Sit all day in front of as many monitors as possible, all with colorful price curves, and wait for the right moment to buy or sell. Apart from the fact that this is arguably the worst job in the world, it offers you exactly two ways of not losing money. Either you know something about a particular asset that all the other traders do not know. Or you have a stroke of luck. The human mind can do a lot, but it can not detect inefficiencies in a price curve, not even with so many help lines. And the problem with the stroke of luck: it can end any time.

Technical analysis. As all studies show, the classic technical indicators such as MACD, Stochastic, Ichimoku etc. are largely useless for trade decisions. However, some indicators, such as the ATR as a measure of volatility, can serve well for auxiliary purposes. And indicators can work for a limited time when the market does not change or when their parameters are constantly adjusted by an algorithm or by real-time optimization. Thus, technical indicators are mostly, but not completely junk.

Elliott waves and Gann squares. Ralph Nelson Elliott claimed in his 1938 book that the prices of financial products would always rise and fall in fractal patterns of five waves. He gave many examples of this. And indeed, you can discover all sorts of waves when you stare at curves long enough. Unfortunately, Elliott forgot to mention any rational way how the markets could produce those waves. Cycles in price curves are quite real and can be used for trading, but do not follow fixed patterns of five or any other number of waves. No systematic studies have so far discovered any trace of Elliott waves in real price curves - nor of the wave patterns by its many imitators.

The Gann lines, squares, and triangles, devised by the luckless trader William Delbert Gann in the early 20th century, didn't fare any better. In his

despair, since he could not support his family with trading, Gann suddenly discovered the path to success: proclaiming himself as a trading genius and offering esoteric trading systems and books. Gann was the ancestor of all scammers in the trading scene. He did not die rich, though, as he apparently lost at the stock market most of the money earned by marketing his methods. But even today many traders believe in his lines, cycles, pyramids and angles.

Astrology. It is not surprising that astrology is widely accepted as a normal trade method and is described in many trading books. How could one resist the temptation to calculate the prices of tomorrow simply from the positions of the sun, moon and planets? Unfortunately, heavenly bodies still refuse to predict earthly events. No backtest has so far revealed any correlation of the full moon with the EUR/USD price. Mercury does not bring the NASDAQ to the rise, and even Saturn can not impress the S & P500 index. Contrary to popular opinion, even the sun is not responsible for seasonal price fluctuations – it's the tilt of the earth axis. So better stay in terrestrial regions.

Japanese candle patterns. With names like 'Three Black Crows' or 'Hidden Baby Swallow', candle patterns bring at least some poetry into the sober trading business. There are about fifty such patterns, mostly three candles with a certain ratio of open, close, high and low, and they are supposed to predict trends. The patterns were invented in the 18th century by Japanese traders from the daily prices of the local rice markets. Possibly, back then they indeed offered a clue for the price of the next day. But even today, many traders are still staring at their price curves in hope of discovering a pattern that promises a profitable trade according to their "Get rich with candle patterns" book.

Of course, other resourceful book authors have invented new patterns with equally impressive names such as 'Lizard's Day' or 'Gilligan's Island'. And when you test them with real price curves other than those used in the book, these patterns yield exactly the same trade result – that is, zero - as the old-fashioned rice-candle patterns. But still, a large number of yet undiscovered patterns can be formed from three or four candles. If your ambition lies there, you can always invent and market them.

Fibonacci numbers. This simple series of numbers - 1, 2, 3, 5, 8, 13, etc. – is often found in systems with geometric growth. But there is no reason to suspect it in the price movements of financial products. And in fact, it isn't there. At least, as far as I know, no one has yet discovered a single property of a price curve in which the Fibonacci series would appear in any form. However, many traders like it, perhaps because the word "Fibonacci" sounds like heavy mathematics. So there are countless Fibonacci trading systems. Whenever a system uses Fibonacci numbers for trade signals, you can safely assume that it would also work, and probably better, with any other numbers.

Harmonic patterns. By connecting certain points on the price curve, you can create funny geometric figures like diamonds, butterflies, crabs, or bats. Their intersections with the price curve tell you precisely the entry points for profitable trades. Or maybe not? Harmonic patterns are certainly profitable - for the software and seminar providers who promote them. But contrary to the claims of these providers, there is no evidence that they ever have helped a trader.

Online analysts. *"The stock XXX was again able to push away from the support line at $ 31.25 on September 17. However, the bearish candle pattern cousels to caution. The further increase of September 20 was neutralized by a Doji candle on the 21st of September Today's third black candle indicates a setback, but if the closing price reaches more than half into the white candle of the 20th of September, an Evening Star pattern would be formed. Therefore the aim of the sellers appears to be the $ 32.50 mark. Positive completion of the pattern above $ 33.30 would counteract further sell-offs."*

This is no spoof - it is the true market valuation of a genuine analyst. Fortunately, such advice is usually free. At least if you do not heed it.

Your trading style. In trading books or seminars, you often get advices such as *Set the stop loss and take profit in a ratio that fits your trading style*. Of course you are wondering what Your Trading Style might be. Do you trade fast, slow, greedy, stingy, or in the style of Kamikaze? And what does this have to do with the stop loss? Nothing at all. Each parameter in a trading system has an optimal value. Any other value thus results in a suboptimal result. As long as it suits your style to win rather than lose, select trade methods and parameters not by style, but by performance.

Your trading plan. You plan to grow your furtune by trading in 3 years from $5000 to $50,000. To do this, you have created a precise plan about how much percent it has to grow each week. Unfortunately, the markets could not care less about Your Trading Plan. If you continue to trade after a loss for keeping the weekly plan, usually only your loss will grow. And the $5000 will grow to $0 - not in 3 years, but considerably faster.

The Holy Grail. On any trader forum you'll find at least one long thread about a system with miraculous profits. The thread starter has discovered the ultimate trade method. He feeds the thread regularly with reports of his impressive results - such as monthly doubling his account - as well as with vague hints about his miracle algorithm and its complex mathematics. His devotees greedily absorb every droplet of information in their endeavor to replicate the trade method - but alas, some vital ingredient is always missing. Miracles have the stupid property of disappearing on closer examination. At some

point, those threads dry out when either the miraculist has gone broke or his last follower has noticed that he was chasing a mirage.

Robots. Theoretically, a correctly developed and reputable robot with a comprehensible algorithm would actually be a good offer. But apparently no one is on sale. You can see in chapter 7 what you're getting instead.

Trading books. There are countless books with countless trade systems - and certainly not all of them are bunk. But surprisingly, most of the praised systems do not even survive a simple backtest. The authors often fear that already. Some warn strongly against testing their system because "backtesting is useless anyway". This is correct insofar as a positive backtest does not prove that the system works. But a negative backtest always means that you will get rid of real money when real trading that system. 90% of the systems from books, forums, or websites that I've tested were clear losers - and that often became apparent in 5 minutes[1].

Trade copy services. The most famous one is Zulutrade®, but there are now many competitors in this lucrative business. The principle is always the same: let others trade for you. Follow successful traders and copy their trades. Theoretically a good idea. If these successful traders really existed there. In fact you seem to have a great selection. On the website of the service, choose one of the top traders with 500% profit and impressive equity curve, invent an amount, and wait for the money streaming in. After a while you'll inevitably notice that it is streaming in the opposite direction. Your top trader had an ugly drawdown shortly after your entry, and his equity curve now looks like the robot curve in Fig. 45. Damn bad luck - or is there something else behind it?

Behind it is no fraud, but simple statistics. There are thousands of traders on those services competing for followers. But the top list only contains those who trade risky and had been lucky so far. Since luck does not last, the

[1] I should mention that the above does not apply to long-term options trading systems or portfolio rotation systems described in books. Those systems, surprisingly, often work. Maybe options trading book authors are more intelligent than other trading book authors?

top traders in the list constantly change. The average survival time in the top list is a few weeks, a maximum of a few months - then the trader's account goes belly up, and with it all his followers. No problem for the trader. After the crash he simply registers again under a new name. He has earned much more with follower commission than lost through trades. Tip: Do not look at the 'Top Traders' but at the 'Top Followers' list, and be aware that the poor performances on that list are in fact the 50 best from ten thousands of followers. This might give you an impression of the likelihood of keeping your money in trade copying.

High accuracy. This means the percentage of won trades. As traders on forums like to brag with their accuracy, here a surefire tip. Simply set the profit target in 5 pips and the stop loss always in 500 pips distance to the current price. You will then be worshipped as god on the forum, at least during your next 99 trades - no matter which strategy you use. Unfortunately the hundredth trade will run into the stop and eat up all profits. Of course, this fatal trade can happen at any time, right at the beginning. Such a method is also often used by gurus in seminars to show live how great they can trade. This can also be achieved with certain exit procedures, a profit target is not absolutely necessary. All these methods have in common that they work a while... until the evil hundredth trade.

Grid Trading. A special case of a system with high accuracy. It subdivides the price range into a regular grid and opens a long and a short trade each time the price crosses one of the grid lines. The trade is then closed with profit when the price crosses the next grid line. This method can generate steady gains for a while, as long as the price moves continuously, but always returns to its original value. Unfortunately, real prices won't do you that favor all the time. If the price moves too far away from the initial value, the money

will be gone[1]. More capital or smaller trade volumes can delay, but not prevent this.

Grid traders are often found in the mentioned copy services, since there it is not important that a system survives for a long time, but that it finds as many followers as possible in the shortest possible time. Only in rare situations grid trading can be useful, when the asset price movements are limited in some way, as was the case with the EUR/CHF price cap. Then the system is protected against fatal runaway prices and can produce steady profits.

Martingale or d'Alembert methods. Increase the stake at each loss. Or - if you do not want to increase the volume, for example for copy services where the lot size is fixed - simply open up a corresponding number of new positions for each lost trade. The theory behind this is that each loss increases your chance to win the next time. Unfortunately, it is not so. On the contrary, inefficiencies often appear or disappear for an extended period of time, so after a lost trade you have a higher chance of losing another. Although martingale systems can make good profit at the beginning - even if trades are opened randomly as in chapter 7 - sooner or later, the unavoidable loss streak[2] will leave the account empty.

---

[1] How much capital do you need for a classic grid trader to survive a price move? Suppose you have **n** open trades and a price movement of **x** pips, where **x** is much larger than the grid distance. The movement will then increase the balance by **x** pips and reduces the equity by approximately **n/2*x** pips, resulting in a total loss of **(n/2-1)*x** pips. On a mini-lot account with 10 lots trade volume, $1 pip cost, and an average of 20 open trades, a price move by 5 cents (500 pips) will create a loss of 9*10*500*$1 = $45,000. Such movements can occur several times a year.

[2] How many trades does a Martingale system survive? Suppose you invested 1% of your capital per trade. After the first loss it is 2%, then 4%, 8%, 16%, 32%, so that after 5 losses the accumulated loss is 63% and the sixth losing trade brings the margin call. At 50% win rate and uncorrelated results, the probability of 6 losses in a row is

Regular income by trading

A winning strategy is only the first step. It is not as easy as it seems to squeeze a steady income out of it. The markets go through periods of varying efficiency and cause the returns to fluctuate strongly. Even systems that generate good returns in the long term have frequent drawdown periods of several months to more than one year. Drawdowns are not a strategy error, but simply a logical consequence of the statistical behavior of price curves.

Clearly, a 12 months drawdown does not fit well along with a regular income. And a drawdown has both psychological and monetary consequences. Do you want to continue a strategy even though it has brought you nothing but losses for months? But if you stop too early, you will be left with the loss. This means that you have to trust a strategy to a certain extent - but only to the loss that is to be expected according to the backtest.

Some rules of thumb to achieve a regular income with algorithmic trading:

Diversification. Do not rely on only one system, trade with as many different methods and algorithms as possible. There is a chance that the drawdown of one system will be compensated by the profits of another. In reality you will notice that most markets are correlated in some way, so that losses can occur in several systems at the same time. Nevertheless, the distribution of your capital to many systems is always preferable and ensures a steady growth.

$p = 0.5^6 = 0.0155625$. The probability that this event does not occur in **n** trades is **$(1-p)^n$**. The average number of trades until the margin call is therefore **$n = \log(0.5)/\log(1-p) = 44$**. With one trade per day, your account will survive about two months. A higher profit rate, such as 90%, does not help - the average loss is usually also correspondingly higher, so the lifetime of your account is still two months. Reducing the investment to 0.1% - that is, $10 per trade at $10,000 capital - extends the average account lifetime to about 17 months.

Enough capital. You should at least double the amount that Zorro displays as "Capital Required". The free Zorro version limits your investment to $ 7000, but you do not need all your funds in the broker's account. You can park it, for instance on some savings account, and invest at the moment when the equity in the trading account drops dangerously. Most brokers allow quick payments by credit card, so you can increase your capital within minutes in dangerous situations.

Solid environment. It is possible, but no good idea, to run a live trading system on your normal working PC. More safe is renting a VPS (Virtual Private Server) to trade. This normally guarantees 24-hour operation and stable Internet connection. You do not necessarily need a special server from a trading colocation provider. Any simple Windows server will do. The cost is around $20 per month. Select the provider well, because you do not want server that is rebooted every few weeks because of system maintenance or updates. Amazon provides servers that run smoothly for years, and do not charge rent for the first year. The installation of a trade system on a server is described in the Zorro manual.

Permanent supervision. Zorro shows your trade status on a web page, which you can install on the server and watch anytime, for instance with a mobile phone. Do this on a regular basis - even during your holiday - and familiarize yourself with the platform, website and telephone number of your broker so that you can intervene quickly in case of an emergency, such as a software crash.

Know the system. You should know how the strategy works and be aware of its risks and returns. Do not trade a system that you do not know or do not understand, and above all, never trade a system that you can not backtest yourself.

Pull out in time. Compare regularly the trade results with the equity curve from the test. If you lose more money than you would expect, do not hesitate and pull the plug. Possibly the market has changed permanently, so that the inefficiency has disappeared. Or something was done wrong in developing the strategy.

At what loss should you stop? The simplest way is the comparison with the normalized 3-year drawdown, the "Capital Required" during the test. The system shall not exceed this loss in the initial period. Afterwards you can use the following formula for an estimate:

$$E = C + P\frac{t}{y} - D\sqrt{\frac{t+l}{y}}$$

Where **E** is the lower limit of your equity, **C** is the initial capital, **P** is the profit achieved in the test, **t** is the live run time of the system, **y** is the test run time, **D** is the maximum drawdown in the test, and **l** is the longest drawdown time. Example: The test yielded $8000 profit after 5 years, with a $2000 drawdown of 1 year duration. You invested $1500 initial capital, of which today only $1000 remain after 6 months losses. The above formula gives E = 1500 + 8000 * 0.1 - 2000 * squareroot(0.3) = $1095. You're below the equity limit. Something is wrong. Pull out.

It should be mentioned that there is a more sophisticated pullout criteria than simple drawdown comparisons: the **Cold Blood Index** (**CBI**) described on the Financial Hacker blog. Zorro displays the CBI on the trade status page whenever you're in a drawdown. So have a look at it from time to time, and keep cold blood.

Don't panic. Fig. 52 below shows the typical equity curve of an algorithmic system traded with real money [1]. From the curve, you can estimate that it has earned about 200% profit in the observed time (750 pips of profit divided by 350 pips drawdown), so it was highly profitable. Nevertheless, it took more than eight weeks for the equity to rise significantly above the zero line. In any real system, the equity fluctuations are always much greater than the upward slope of the equity curve. Most successful strategies are losing for up to 90% of their time (meaning that the equity is almost always below a previous high). For this reason, as you can see in Fig. 46, your account balance is normally below the initial capital in the first few weeks after the start of the strategy. If this were not the case, you would not need any investment capital aside from a small amount for the margin.

[1] It was a grid trader exploting the Swiss Franc cap.

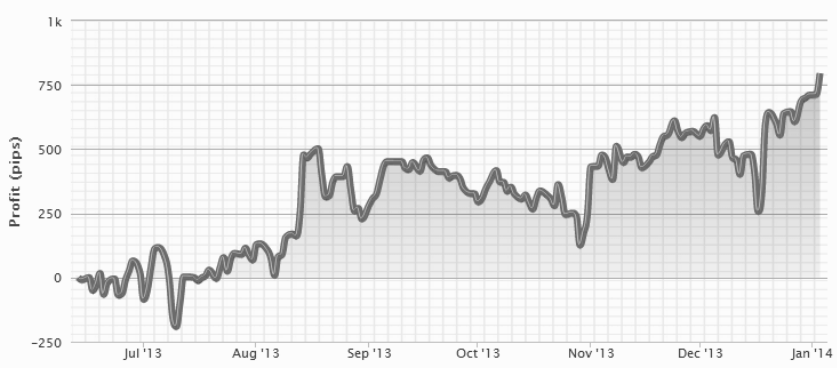


Fig. 52 – Bumpy live start (screenshot from ZuluTrade™)

If you panic on seeing such a drawdown and stop the strategy, the money is gone. This way beginners often manage to lose money even with a highly profitable system. Keep in mind that there were long and dreary drawdowns in the test and that the simulation still ended with a nice profit. These drawdowns will also be encountered in real trading. Admittedly, a series of 20 consecutive losses with real money is a special experience. Bite your teeth and sit it out. At least as long as the equity has not fallen below the limit according to the above exit formula.

Do not interfere. It's hard to see winnings dwindle in a drawdown. But never forget that the strategy, if correctly developed, is already optimized (or at least should be). Any intervention in the system - especially manually closing or keeping of trades - thus will reduce the expected profit. Tampering with the system makes sense only if you know something the strategy does not know, for instance that a stock market crash is imminent. Otherwise, leave it even if the temptation is so great. For the same reason, do not stop and restart a system all the time. Any early closing of trades intervents with the algorithm at the price of several hundred pips, and can turn a winning system into a loser.

Do not get greedy. If your winnings accumulate, you will be tempted to raise the stake. Think about the square root rule. If the account balance has doubled, only increase it by 40%. Otherwise, stressful times will come when a drawdown begins, the money buffer on your account becomes dangerously thin and you have to deposit hastily. This usually happens on holidays.

Although highly profitable strategies are often rather simple, it is important to avoid the many pitfalls not only during their development, but also during the test and during live trading. Using algorithmic systems to achieve a steady income from which you can live is no easy task. Otherwise, anyone would do

it. You can always rely on one thing: the future is unknown. What worked in the past does not necessarily work in the future. Do not trust on your luck, trade only with the money that you can afford to lose, and always have a plan B, C, and D. Learn as much as you can and let as many different strategies as possible work for you. Then there's nothing in your way to financial freedom.

Appendix

The Market Meanness Index

Every technical trader has the sacred duty of inventing at least one new indicator in his lifetime and adding it to the long list of existing indicators. So here's my modest contribution, the Market Meanness Index (MMI), which was used by Alice in the third chapter to improve her trend following system. The source code of the MMI is not too complex:

```
var MMI(vars Data,int TimePeriod)
{
  var m = Median(Data,TimePeriod);
  int i, nh=0, nl=0;
  for(i=1; i<TimePeriod; i++) {
    if(Data[i] > m && Data[i] > Data[i-1])
      nl++;
    else if(Data[i] < m && Data[i] < Data[i-1])
      nh++;
  }
  return 100.*(nl+nh)/(TimePeriod-1);
}
```

As the name implies, the indicator measures the mean reversion tendency of the market – the tendency of the price curve to start a trend in some direction, but then return to the old value. When it happens frequently, it causes all trend following systems to bite the dust.

A series of random numbers reverts to the mean – or more precisely, to the median – with a probability of 75%. So when you look at a random number sequence, if the previous number was above the median, in 75% of all cases the current number will be lower than the previous one. And if the previous number was below the median, 75% chance is that the current number is higher. This is valid for all numbers in the sequence. The proof of the 75% is relatively simple and won't require integral calculus. Consider a sequence of numbers – say, a sequence of prices - with median **M**. By definition, half the prices are less than **M** and half are greater (for simplicity's sake we're ignoring the case when a price is exactly M). Now combine the prices to pairs

each consisting of a price Py and the following price Pt. Thus each pair represents a price change from Py to Pt. We now got a lot of price changes that we divide into four sets:

1. (Pt < M, Py < M)
2. (Pt < M, Py > M)
3. (Pt > M, Py < M)
4. (Pt > M, Py > M)

These four sets have obviously the same number of elements – that is, 1/4 of all Py->Pt price changes – when Pt and Py are not correlated, i.e. completely independent of one another. The value of **M** and other properties of the price sequence won't matter for this. Now how many price pairs revert to the median? All pairs that fulfill this condition: (Py < M and Pt > Py) or (Py > M and Pt < Py) The condition in the first bracket is fulfilled for half the prices in set 1 (in the other half is Pt less than Py) and in the whole set 3 (because Pt is always higher than Py in set 3). So the first bracket is true for 1/2 * 1/4 + 1/4 = 3/8 of all price changes. Likewise, the second bracket is true in half the set 4 and in the whole set 2, thus also for 3/8 of all price changes. 3/8 + 3/8 yields 6/8, i.e. 75%. This is the three-quarter rule for the differences of random numbers.

The MMI function above just counts the number of data differences for which the conditition is met, and returns their percentage. The **Data** series may contain prices or price changes. Prices have always some serial correlation: If EUR/USD today is at 1.20, it will also be tomorrow around 1.20. That it will end up tomorrow at 70 cents or 2 dollars per EUR is rather unlikely. This serial correlation is also true for a price series calculated from random numbers, as not the prices themselves are random, but their changes. Thus, the MMI function should return a smaller percentage, such as 55%, when fed with prices.

Unlike prices, price changes have not necessarily serial correlation. A one hundred percent efficient market has no correlation between the price change from yesterday to today and the price change from today to tomorrow. If the MMI function is fed with perfectly random price changes from a perfectly efficient market, it will return a value of about 75%. The less efficient and the more trending the market becomes, the more the MMI decreases. Thus a falling MMI is a indicator of an upcoming trend. A rising MMI hints that the market will get nastier, at least for trend trading systems.

One could assume that the MMI has some power to predict the price direction. A high MMI value indicates a high chance of mean reversion, so when

prices were moving up in the last time and MMI is high, can we expect a soon price drop? Unfortunately it doesn't work this way. The probability of mean reversion is not evenly distributed over the length of the **Data** interval. For the early prices it is high (since the median is computed from future prices), but for the late prices, at the very time when MMI is calculated, it is down to just 50%. Predicting the next price with the MMI would work as well as flipping a coin.

Useful books

Not all books about trading methods and systems are junk. Some are in fact useful. Here is an incomplete list of useful books for algorithmic trading:

Jaekle / Tomasini: Trading Systems. How to develop a strategy; the book guides you through every step from the idea through to various optimization and money management methods. However, better do not really trade the system presented therein!

David Aronson, Evidence-Based Technical Analysis. An excellent, if somewhat wordy book on testing trading systems. Contains the first comprehensive test of the usefulness of traditional indicators.

Ernest P. Chan, Quantitative Trading. Testing strategies and portfolio optimization with lots of practical tips.

John F. Ehlers, Cycle Analytics for Traders. Algorithmic trading with new methods of signal processing. With source code for all described algorithms.

Ralph Vince, The Handbook of Portfolio Mathematics. How to reinvest your capital and divide it between different assets and strategies.

William R. Gallacher, Winner Take All. An intelligent and witty insight (from 1994) into the trading scene and their gurus. The author also describes a seemingly profitable trading system and then shows why it does not really work - unique to a trading book.

Robert Harris, The Fear Index. Mandatory for all developers of trading systems.

Script download

All strategies, as well as the scripts for generating the figures and diagrams in this book, can be downloaded from **https://financial-hacker.com**. Zorro version 2.08 or above is required for running the scripts. The zip password is "Bob".

About the author

Johann Christian Lotter has a degree in physics and is CTO with oP group Germany, a software firm that firstly produced movie effects and computer games in the 1990s. Now they are developing algorithms for financial analysis and machine learning. So far he and his team have programmed more than 900 trading systems for institutions and private traders. In his free time he enjoys writing mystery novels and undertaking travels to remote areas.