Genetic Programming with Syntactic Restrictions applied to Financial Volatility Forecasting

Gilles Zumbach[1], Olivier V. Pictet[2], and Oliver Masutti[1]

Abstract

This study uses Genetic Programming (GP) to discover new types of volatility forecasting models for financial time series. GP is a convenient tool to explore the space of potential forecasting models and to select the more robust solutions. The application to foreign exchange financial problems requires an exact symmetry induced by the interchange of currencies. In GP, this symmetry is enforced by using a strongly typed GP approach and syntactic restrictions on the node set. GP convergence is increased by a few orders of magnitude by optimizing the constants in the GP trees with a local optimization algorithm. The various algorithms are compared on the discovery of the symmetric transcendental function cosine. For the volatility forecast, the optimization is performed using return time series sampled hourly, possibly including aggregated returns at longer time horizons. The in-sample optimization and out-of-sample tests are performed on 13 years of high frequency data for two foreign exchange time series. The out-of-sample forecasting performance of these new models are compared with the corresponding performance of some popular ARCH-types models, and GP consistently outperform the benchmarks. In particular, GP discovered that cross products of returns at different time horizons improve substantially the forecasting performance.

Key words: *Volatility forecasting, Genetic Programming, Syntactic restrictions*

[1]Olsen & Associates
Research Institute for Applied Economics
Seefeldstrasse 233, 8008 Zürich, Switzerland.
e-mail: gilles@olsen.ch
phone: +41-1/386 48 48 Fax: +41-1/422 22 82

[2]Dynamic Asset Management
Chemin des Tulipiers 9, 1208 Genève, Switzerland.
e-mail: opictet@dam.ch

1 Introduction

One challenge posed by the financial markets is to correctly forecast the daily volatility of financial assets in order to obtain reasonable predictions of the potential risks, or to optimally allocate assets in a portfolio. Clearly, the volatility is itself a non trivial process with interesting dynamics, and from the analysis of empirical data, a large number of stylized properties related to volatility are known. The most important of these properties is the long memory of the volatility, as measured by a lagged autocorrelation function that decays as a power law. This property is also called volatility clustering, and the slow decay of the autocorrelation means that this clustering is present at all time horizons. The simplest model that describes volatility clustering is the GARCH(1,1) model. Yet, this model has an exponential autocorrelation function for the volatility, meaning that it captures the volatility clustering only at one time horizon. In order to remedy the different shortcomings of the simplest GARCH(1,1) model, a large number of variations in the popular ARCH (Autoregressive Conditional Heteroskedasticity) class of models has been studied, mostly using daily data. The use of data at higher frequency opens new avenues in volatility forecasting as the statistical uncertainty is decreased and intra-day effects must be taken into account. But consequently, the complexity of the problem increases. Undoubtedly, the use of high frequency data permits better short term volatility estimations, but also implies more complexity in data treatment and volatility modeling. The main problem can be summarized as: what are the most important stylized properties that must be taken into account in order to obtain a good volatility forecast? In this study, we use Genetic Programming (GP) to discover new types of volatility forecasting models for foreign exchange (FX) rates at hourly frequency.

Genetic Programming is a tool that searches in the space of possible programs, represented by trees, for individuals that are fit for solving the given problem [Koza, 1992]. For the application of GP techniques to financial data, a similar approach was recently applied with some success on trading model discovery [Bhattacharyya et al., 1998, Chopard et al., 2000], and we use it in this paper to explore the space of volatility forecasting models. In the application to foreign exchange financial time series, there is an important exact symmetry induced by the exchange of the two currencies, and this symmetry must be respected by the solutions. For this purpose, we have used a strongly typed GP approach, where the typing system keeps track of the parity of the GP trees. In this way, we have reduced our search space from all possible GP trees down to the subspace of trees that have the proper symmetry.

From a general point of view, volatility forecast is a function fitting problem, where the realized value for the volatility is the function to be discovered using a causal information set. The time series of the realized volatility is dominated by randomness, and the actual amount of information contained in the information set about the future evolution is rather low. This is measured for example by the lagged correlation for the volatility, which is in the order of 3 to 15%, depending on the actual definition of volatility. In short, this means that the volatility forecast is a very difficult challenge for GP, and the algorithms need to be very efficient. In order to develop and test tools, we have applied GP to a similar problem, namely the discovery of the transcendental function cosine, using polynomials. As for the volatility, the cosine function obeys a symmetry of parity $\cos(-x) = \cos(x)$. The study with the cosine makes clear that the conventional GP cannot tackle the volatility forecast problem, and the main difficulty lies in the discovery of good values

for the constants included in the GP trees. In order to speed up convergence to the optimal values of the constants, we have to use a local search algorithm, like a conjugate gradient. Only when using a mixed algorithm, are we able to obtain good solutions for the cosine problem, and to find volatility forecast that can compete with the standard GARCH(1,1) model.

This article is organized as follows. In section 2, the Genetic Programming approach is described. A brief introduction to GP is given is the subsection 2.1, followed by the GP with types and the related syntactic restrictions in subsection 2.2. The modifications to the evolution operators required by the GP with types are discussed in subsection 2.3. Subsection 2.4 presents how to use GP with types to impose a particular symmetry on the solutions. In subsection 2.5, we give the set of nodes we have used, together with their respective syntactic restrictions. Control over the actual CPU time is critical for the application on volatility forecasts, and this means controlling the complexity of the generated trees, as discussed in the subsection 2.6. The section on GP itself concludes with the combination of GP with with local search techniques for the optimization of the tree's constants. The section 3 discusses the application of GP to the discovery of the cosine function by polynomial approximations. This includes a detailed comparison of the various algorithms, as well as optimization of the evolution operators used by the GP. The application to volatility forecast is presented in the section 4. The peculiarities of the financial volatility are presented in the introductory part 4.1, and the daily volatility forecast in section 4.2. The volatility processes used as benchmark for comparison with the performance of GP trees are given is the section 4.3. Section 4.4 presents the results of GP search for volatility forecast, and conclusions are drawn in section 5.

2 Genetic Programming with Syntactic Restrictions

2.1 Introduction

Genetic programming [Koza, 1992] is a tool to search the space of possible programs for an individual (computer program) that is fit for solving a given task or problem. It operates through a simulated evolution process on a population of solution structures that represent candidate solutions in the search space. The evolution occurs through:

- a selection mechanism that implement a *survival of the fittest* strategy
- genetic *cross-over* and *mutation* of the selected solutions to produce offspring for the next generation.

The generated programs are represented as trees, where nodes define functions with arguments given by the values of the related sub-trees, and where leaf nodes, or *terminals*, represent task related constants or input variables.

The selection mechanism allows random selection of parent trees for reproduction, with a bias for the trees that represent better solutions. Selected parents are either mutated, or used to generate two new offsprings by a cross-over operator. Cross-overs and mutations are the two basic operators

used to evolve a population of trees. The mutation operator effects random changes in a tree by randomly altering certain nodes or sub-trees, whereas the cross-over operator is an exchange of sub-trees between two selected parents. The evolution of trees' populations continues until a certain stopping criterion is reached. The initial population is composed of random trees, which are generated by randomly picking nodes from a given *terminal set* and *function set*. The only constraint is that the generated trees ought not to be too complex. A restriction on the maximum allowed depth or the maximum number of nodes is also frequently imposed.

Since the evolution operators in GP can establish arbitrary functions and terminals as arguments (descendants) for a function node, the function set is required to be closed with respect to the various arguments that it can have. The closure hypothesis states that the result of any node can be used as argument of any node, and this permits easy implementation of the cross-over and mutation operators. Yet, this leads to programs with unnatural structures, like boolean operators taking real arguments, or multiplication nodes operating on the results of a boolean operation.

Nowadays, modern programming languages support the notion of types, and it is quite natural to introduce this concept into GP trees. This study follows the strongly typed GP approach [Montana, 1995], where the branches of a tree carry a *data type*, and the operators (nodes) accept only some combination of types. Yet, the typing system is used in this paper to impose an exact symmetry on the generated programs. This symmetry is induced by a parity transformation, namely in the foreign exchange market, by the permutation of the currency pair. In technical terms, the typing system is used to impose a well defined representation of the solution in the Z_2 group generated by the graduation of the problem by a parity transformation.

2.2 Typing system and syntactic restrictions

A genetic program is a tree, whose nodes represent functions (with its subtree as function arguments), and whose leaves are associated with task-related data input or "constants". As the tree-node-terminal terminology is not always accurate and rich enough, we also use a function terminology. The *n*-arity of a function fixes the number of subtrees attached to the corresponding node, while a function without argument is a terminal node:

$$f^{(n)} \; : \; \mathbb{R}^n \longrightarrow \mathbb{R}.$$

In GP, it is usually postulated that *any* tree is a valid program (the closure property). The only restriction is the *n*-arity of the functions, which fixes the topology of the tree. In order to introduce types, a set of *N type* C_i is defined. A simple example with two types is C_1 = boolean and $C_2 = \mathbb{R}$. Then, a function is a mapping from types into a type, namely

$$f^{(n)} \; : \; C_{i_1} \times \cdots \times C_{i_n} \longrightarrow C_{i_{n+1}} \quad i_j \in 1, ..., N. \tag{1}$$

For example, with the two above types, we can have the functions of two arguments:

$$\begin{aligned} > \quad &: \quad \mathbb{R} \times \mathbb{R} \rightarrow \text{boolean} \\ + \quad &: \quad \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}. \end{aligned}$$

The typing of the functions induces *syntactic restrictions*, namely valid programs are trees where, for all the branches, the output type of the argument functions are valid argument types for the function node. The set of all programs that obey the syntactic restrictions defines the space of programs with types, with a possible further constraint on the type of the root. Notice that these syntactic restrictions correspond exactly to the syntactic restrictions of any strongly typed language, like C++.

As soon as the types of the variables in a problem are not unique, like for example with the two types boolean and real, it is indeed very natural to introduce such syntactic restrictions. This avoids the rather unpleasant and unnatural condition that a function must be applicable to all types of argument. In 'traditional' GP, the closure property imposes an embedding of the types into $\mathbb{R}$. For example, booleans are represented by a real number, with the value zero for false and different from zero for true. This embedding allows free mixing of the types. Yet, if the types become more complex, for example with vectors, matrices or lists, this embedding becomes completely artificial, if not impossible. Therefore, the introduction of a proper typing system is a mandatory step toward the construction of more complex random programs. For the present paper, our use of typed GP is somewhat different as we use it to impose a given symmetry of parity to the solution. This kind of use is possible only with types.

With the introduction of types, the problem to be solved is to find proper evolution operators, namely to generate random trees, mutations and cross-over operators that respect the syntactic restriction. In other words, the evolution operators must be involutions in the space of typed GP. Moreover, they must be ergodic on the space of typed programs. Beyond ergodicity, the cross-over and mutation operators must also be 'efficient' on the space of programs, and their efficiency may be related to the number of types and functions.

From the point of view of the implementation, the types are simply a mapping to a set of integers, namely, to each type corresponds a unique number. The specification of the functions must include the possible types of their arguments and the type of the results. Then, the syntactic restrictions correspond simply to the equality between numbers across connected nodes and terminals. It is therefore quite simple to implement an arbitrary number of types and the corresponding syntactic restrictions. The implementation of the evolution operators is more complex, in particular some operations might fail and a recovery mechanism must be implemented.

2.3 The evolution operators

The following section describes the elementary operations used to implement the evolution operators. Because of the types, the set of possible operations is larger than in the usual GP approach.

2.3.1 Construction of a random tree

Genetic program trees are grown by recursively appending nodes or leaves that match the type restrictions of the parent nodes to the dangling connections. The complexity of the tree is controlled by biasing the probability to return a node or a terminal, depending on the depth of the current

connection compared to a target depth. If a maximal depth is exceeded, the attempt to construct a random tree fails.

2.3.2 Mutation operators

The mutation operations we implemented for the genetic program are the following.

- Constant leaf mutation
  A constant terminal is mutated by either changing the sign of the constant (with a probability of 10%), or by multiplying the constant value by e^x, where x is a random number uniformly distributed between $-\ln(\sqrt{2})$ and $\ln(\sqrt{2})$.

- Node substitution
  A node is replaced by another, while the type constraints are preserved.

- Subtree mutation
  A branch is picked at random, the tree attached to this branch is deleted and a new random tree with the same root type is attached to the branch.

- Branch type mutation
  A node is picked at random and is checked to see if a branch type can be changed while keeping the type for the other branches. If this type change is possible, the sub-tree attached to this branch is deleted and a new random tree with the new selected root type is attached (if several new types are possible, one is chosen at random). This particular mutation is important as it allows us to replace a constant terminal by a subtree.

- Root splicing
  A node is inserted as the new root node of the tree (i.e. on top of the existing root node), and the possible dangling branches are filled with new random subtrees. Again, the type constraints have to be respected, in particular with a possible restriction of the root type.

- Node insertion
  A node is inserted randomly by picking a branch at random, inserting a random node compatible with the branch type, and completing the possible new node's dangling branches with random trees.

- Node deletion
  A node is selected randomly for deletion. If an argument's type matches the type of the results, the node is deleted and the sub-tree attached to the parent, while the other possible sub-trees are deleted. If no descendant can be attached to the parent of the selected node (because of type incompatibility), another candidate node is selected for deletion. After a fixed number of failed attempt, node deletion fails.

  This mutation tends toward simpler trees, and balances the 'node insertion', 'root splicing' and 'branch type mutation' which tend to create more complex trees.

In these various operators, when a random sub-tree must be generated, the global depth of the tree is controlled by the same mechanism used to grow random trees.

2.3.3 Crossover operator

The crossover operator takes two genetic programs as arguments. It selects a branch randomly in the father tree, and checks if there exists at least one branch with matching type in the mother tree. If yes, a branch with the selected type is chosen randomly in the mother tree and the two selected subtrees are swapped. If not, the above procedure is repeated. If the crossover procedure still fails after a maximal number of attempts, the crossover operation fails.

2.3.4 Markov chains parametrization

We have defined 7 kinds of mutation, and when a given tree is selected for mutation, one of these operators is randomly chosen according to some specific predefined selection probabilities. The probabilities of selecting each particular mutation parametrize the mutation operator on the GP space. The genetic algorithm also performs cross-overs with a given probability. All these probabilities can be given in a vector that parametrizes the Markov chains, namely

$$\begin{aligned}\vec{p}_{\mathrm{MC}} \;=\; \{\;& p(\text{constant leaf mutation}),\\ & p(\text{node substitution}),\\ & p(\text{subtree mutation}),\\ & p(\text{branch type mutation}),\\ & p(\text{root splicing}),\\ & p(\text{node insertion}),\\ & p(\text{node deletion}),\\ & p(\text{cross-over})\;\}\end{aligned}$$

with the constraint that the sum over the first 7 values must be one. In section 3, this vector of probabilities is optimized using a genetic algorithm in order to have the most efficient genetic evolution.

2.4 Using the typing system to impose a symmetry on the solution

In some applications, the generated solutions must respect some specific symmetry conditions inherent in the problem to be solved. The main example presented in detail in section 4, is the case of the volatility forecast models based on price changes r. The volatility must be identical for a specific rate and for the corresponding inverted rate, i.e. $\sigma(-r) = \sigma(r)$. A simpler example, which is used as a test case for this study, is the fit of symmetric functions, like cosine, with polynomials approximations (see section 3). In such an application, the goal is to obtain GP trees which represent good approximations to the original symmetric functions and with the same

symmetry properties. The output of the GP trees is required to be symmetric $T(-x) = T(x)$ and we use the syntactic restrictions to impose the symmetry of the solution.

In order to obtain meaningful solutions with genetic programming, we need to generate GP trees which returns a symmetric output at the root node. The symmetry type at each node of the GP trees need to be evaluated to enforce the overall symmetry properties. This is feasible if the symmetry of each component of the tree is classified according to its symmetry property. All the functions or terminals used in the construction of a GP tree are classified according to the three output *types*:

- antisymmetric type A: $A(-x) = -A(x)$ (e.g. x or x^3),
- symmetric type S: $S(-x) = S(x)$ (e.g. $|x|$ or x^2),
- constant type C: numerical constants, not affected by the transformation $x \to -x$.

A simple example shown in figure 1 is a GP tree corresponding to the polynomial $T(x) = a + x * (x * b) = a + bx^2$. Each branch has been labeled according to its type (A, S or C). The types are determined starting from the terminals and moving upward to the root using the parity properties of each node. When growing random trees or modifying trees, the types of the branches have to be computed, in order to check that the tree is valid (i.e. obeys the syntactic restrictions) and returns the requested type.

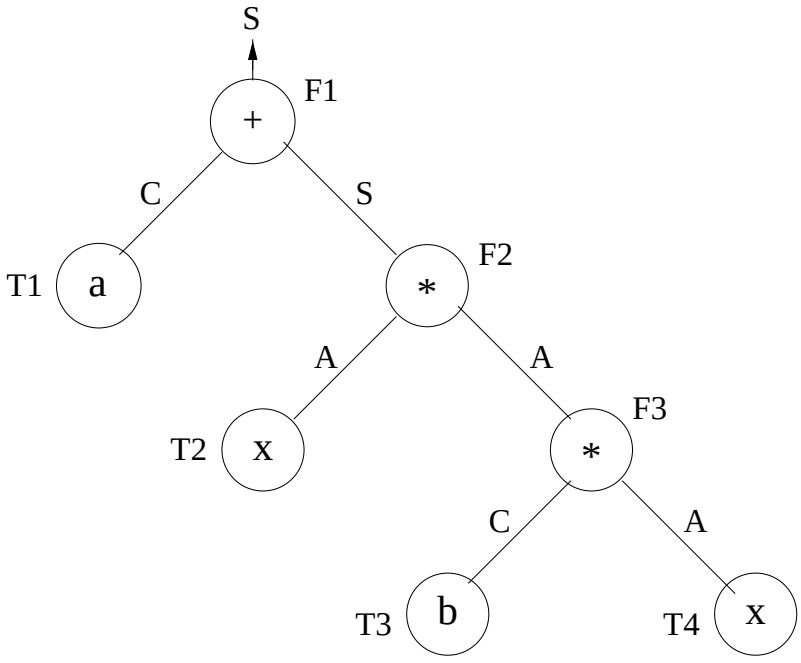

Figure 1: GP tree with symmetric output

2.5 The nodes, with their syntactic restrictions with respect to parity

In the construction of new random trees, and in the cross-over and mutation operators, the possible combination of branch types allowed for a specified function are given in the *syntactic restriction tables*. Such tables must be provided for each function, and are given below for the operators we have used. Most of the operators take two arguments. In the tables below, the row and column labels correspond to the type of the two arguments, with the intersections providing the type of the result. The symbol "–" represents a combination of arguments which is not allowed. In the syntactic restriction tables, operations between constants are disallowed, to avoid the wasteful computation of constants through the use of cross-over and sub-tree mutation operators. This will forbid trees like $(a*b)*x$. Yet, this is not enough to remove completely redundant constants, as for example in the tree $a*(b*x)$.

The elementary algebraic functions have straightforward properties with respect to the parity transformation. The syntactic restriction table for the addition and subtraction operators is

+ –	A	S	C
A	A	–	–
S	–	S	S
C	–	S	–

and for the multiplication $(*)$ and division $(/)$ operators it is

* /	A	S	C
A	S	A	A
S	A	S	S
C	A	S	–

For the volatility forecast models, more operators are used in the construction of GP trees. We have taken a function set composed of the simple arithmetic operations $\{+, -, *\}$ and some operators commonly used in volatility processes such as exponential moving averages, square and absolute value functions.

The exponential moving average operator EMA$[x]$ evaluates by a simple iterative formula the average value of x on a moving sample

$$\mathrm{EMA}(t) \;=\; \mu\,\mathrm{EMA}(t-\delta t) + (1.0-\mu)\,x(t) \tag{2}$$

with $\mu = \exp(-\delta t/\tau)$ and $0 < \mu < 1$. The range $\tau > 0$ of the EMA gives the length of the moving sample. A convenient parametrization of the range τ is given by the logarithm of the range $z = \ln(\tau)$. The use of logarithm of the time range is natural in problems where τ can vary widely, from hours to months. Moreover, for the GARCH(1,1) volatility model, z has been shown to give an efficient parametrization of the model for the optimization, with the further advantage of having no domain restriction [Zumbach, 2000a]. Therefore, our EMA node takes as first argument z, and as second argument x. The syntactic table with respect to parity is

EMA	A	S	C
A	–	–	–
S	A	S	–
C	A	S	–

where the row label corresponds to the type of z, and the column label is the type of the variable argument x to be averaged. As this operator is an averaging operator, the type of the result is always the type of the argument x. The absolute value and the square operators take one argument (A or S) and always return a symmetric type (S) value.

2.6 Taming complexity

One important problem in genetic programming is the control of complexity. Complexity is not bad in itself as it can be a reservoir for mutations. Yet, more complex trees take longer to evaluate, and our Markov chains have no problem in generating more complex trees when needed. Therefore, to keep the CPU time low, complexity must be tamed down. For the volatility forecast inference, a too complex tree can also overfit the data set, namely a solution can fit particularly well the sample of data used for the GP optimization (in sample) but can fail on another sample (out of sample). This overfitting problem is particularly acute for financial applications, like volatility forecast, as the dominant character of the target function is random. The number of constants in the GP tree is particularly critical as too many constants imply a strong overfitting probability and a very slow convergence of the local optimizer (see next section). For these reasons, we have adopted an approach where we correct the score function to penalize more complex GP trees.

There is no natural measure of the complexity of a GP tree, and several definitions can be written down. We have used a simple approach using the number of nodes (including the terminal nodes) and the number of constants in a GP tree (the maximal depth can also be used). The complexity of a GP tree is defined as

$$\text{complexity} = \sum_k w_k \, \exp(f_k/r_k) \qquad \text{with} \quad \sum_k w_k = 1 \tag{3}$$

where the sum runs over the factors f_k (number of nodes and number of constants), w_k is a weight, and r_k the range at which the complexity measured by the factor k starts to increase. For the runs below, we have used the parameters $w = 0.2$, $r = 20$ for the number of nodes, and $w = 0.8$, $r = 4$ for the number of constants.

For both applications below, the score is given by an L^2 distance, which is always positive. Close to the solution, we need to enhance small differences in the score in order to put a strong enough selection pressure on the best trees. A natural solution is to select the GP trees according to the logarithm of the score. With the penalization of complexity, we have used a fitness given by

$$\text{fitness}_i = \log(\text{score}_i) + \lambda \, \text{complexity}_i \tag{4}$$

where the index i runs over the GP trees in the population. The parameter λ fixes the importance of the complexity penalty. This parameter has been optimized (see section. 3), and values in the range $\lambda \simeq 1$ lead to an efficient algorithm.

For the genetic evolution, trees are selected using a fitness proportional algorithm. The probability of selecting an individual depends then on an overall additive constant in the fitness, for example on a change of normalization in the L^2 distance used in the score. In order to fix this additive constant, we use

$$p_i = \max(-\text{fitness}_i + C, 0) \tag{5}$$

where p_i is the probability to select the individual i. The minus sign originates in that the best fits have the lowest value, but need to be selected with the highest probability. The constant C is fixed by giving the probability of selecting an individual among a given best fraction, for example, we want to select with a probability of $p_b = 50\%$ an individual among the best fraction $b_f = 25\%$ of the population. A simple computation gives

$$C = \frac{\sum_{i \leq b_f n} \text{fitness}_i - p_b \sum_{i \leq n} \text{fitness}_i}{b_f n - p_b n} \tag{6}$$

with n the population size. This method can lead to negative probability for the worst individuals, and in such case, the negative probabilities are replaced with 0. For the computations below, we have used $p_b = 50\%$ and $b_f = 25\%$, namely half of the time, a tree in the best 25% is selected.

2.7 Combination with local search techniques

As is usually done in GP, the genetic algorithm also optimizes the constants included in a GP tree by random mutations. From the algorithmic point of view, this optimization of the constants is very inefficient. Moreover, for the volatility forecast modeling, the forecast performance depends critically on having good values for the constants. For the simple GARCH(1,1) model, the standard optimization (using the derivatives) of the model parameters is already a difficult problem in itself [Zumbach, 2000a]. When exploring various models, the selection between two good volatility models is in fact based on very small differences between the score values and a small change in one of the constants generally alters a near optimum solution to the average ones.

In order to avoid such problems and to retain the optimum solutions corresponding to each possible structure of new volatility forecast models, we combine the genetic programming search with a local optimizer. It must be noted that an optimization algorithm using a gradient (like a simple conjugate gradient) is far superior to a random search when optimizing a sufficiently smooth function in $\mathbb{R}^n$. Therefore, the constants in a genetic program can be optimized by having the genetic program evaluation function call a local optimization algorithm that searches for the best constants, and inserts these values back into the tree. The local optimization algorithm finds the best constants, starting from the values given in the original tree. The gradient of the cost function needs to be evaluated numerically, and we use the BFGS algorithm for the local search [Press et al., 1986]. In this way, both genetic programming and the local optimization algorithm are used in the domain where they perform best, namely GP to explore the structure of the solution, and the local optimizer to adjust the parameters to the problem. The reader should be warned that the function over $\mathbb{R}^n$ generated by random trees can have quite wild shapes, and that the local optimization algorithm must be very robust to be able to gracefully handle thousands of searches. In the next section, we compare GP with and without local optimization of the constants.

3 Function fitting

We have tested our GP algorithm on the discovery of the transcendental function $\cos(x)$. The "residual error" is given by

$$d^2 = \left(\frac{2}{n} \int_0^{2n\pi} dx \left(\cos(x) - \mathrm{gp}(x)\right)^2 \right) \tag{7}$$

with $\mathrm{gp}(x)$ the value of a GP tree with the single argument x (i.e. the terminal nodes can have the value x or a constant). The integer parameter n fixes the number of cosine periods to include in the integral, and the normalization of the integral is chosen so that for $\mathrm{gp}(x) = 0$, the cost function is 1 (regardless of n). For the genome without syntactic restrictions, namely when not imposing the symmetry of parity, we have also measured the convergence properties with the integral defined on the symmetric interval $[-2n\pi, 2n\pi]$ (with the normalization constant $1/n$). The cost function to optimize by GP is the *logarithmic residual error*

$$\mathrm{score} = \log(d) \tag{8}$$

with log the base 10 logarithm (for the computation of the fitness, eq. 4 includes the score directly and not its logarithm). The node set includes the function +, * and square, so that the search is done in the set of polynomial functions. Practically, the above integral is discretized with a step $dx = 1/32$.

The discovery of transcendental functions by GP is an interesting test bed for several reasons:

- It is a non trivial task, and the difficulty can be increased, in the present case by increasing the number of periods n. The parity of the function to discover can be given, in our case $\cos(x)$ is an even function of x. Therefore, the influence of the syntactic restrictions can be measured. Notice that with the syntactic restrictions, the fit is done implicitly on the interval $-2n\pi$ to $2n\pi$, whereas without restrictions the fit is done only on the interval included explicitly in the above integral, namely from 0 to $2n\pi$ or from $-2n\pi$ to $2n\pi$.
- The computational time for one search is small enough to allow us to perform statistics with various algorithms. Moreover, with good average values, the algorithm can be optimized. For example, we can use a genetic algorithm to optimize the parameters of the Markov chains (mutation and crossover) used by the genetic program.
- When searching within the set of polynomial functions, the irreducible polynomial corresponding to a GP tree can be computed. This is interesting for two reasons. First, the irreducible polynomials provide for equivalence classes of GP trees, where each equivalence class is characterized by the maximal order of the Taylor expansion of the cos function. For the algorithm with local optimization, the irreducible polynomial coefficients are unique in each equivalent classes (up to numerical convergence error) and correspond to the best approximation of the cosine function by a polynomial of a given order. Second, a polynomial space has a simple metric, whereas GP trees have no obvious metric. As the algorithm converges to the solution, the irreducible polynomial must converge to the Taylor expansion of the chosen function, and this distance can be measured easily.

The Markov chains on the tree space have been optimized on this problem. For this purpose, an auxiliary cost function is set, with value given by the average of the score eq. 8 over 20 GP runs. The actual CPU time for the optimization is also measured, and the final parameters chosen so that the computational time stays small as well. A genetic algorithm is then used to optimize the probabilities for the mutation and cross-over $\vec{p}$, as defined is sec. 2.3.4, as well as the weight for the complexity λ in eq. 4 The main algorithms to consider are the GP with or without syntactic restrictions, and with or without local optimization of the constants. The optimization has been done on the 4 combinations of syntactic restriction and local optimization. Essentially, the efficiency of the Markov chain is changed dramatically by local optimization, but not by the syntactic restrictions.

With local optimization, the probability to mutate a constant is set to zero, and the GP terminates after 50 generations. The results are as follow: the "node substitution" is an efficient transformation; the "branch type mutation" is efficient, whereas the "subtree mutation" is not; the "root splicing" is counter productive; the "node insertion" is efficient; the "node deletion" should be used parsimoniously; the cross-over probability is found to be fairly irrelevant. Quantitatively, a good vector of probabilities characterizing the Markov chain is $\vec{p}_{\mathrm{MC}} = (0.0, 0.4, 0.2, 0.05, 0.0, 0.25, 0.1, 0.5)$. These values have been used for the comparison below, as well as for volatility forecasting in the next section. Without local optimization, the probability to mutate a constant is also optimized, and the GP run terminated after 5000 generations. The results are similar to those above, but with the "constant leaf mutation" being very important. The vector of probability is set to 0.5 for the "constant leaf mutation", and half of the values above for the mutations, namely $\vec{p}_{\mathrm{MC}} = (0.5, 0.2, 0.1, 0.025, 0.0, 0.125, 0.05, 0.5)$.

Figure 2 displays the mean logarithmic residual error as a function of the number of generations. The advantage of the algorithm with local optimization is striking (let us emphasize that the horizontal axis is logarithmic). Yet, this comparison is unfair as the computation time needed for the optimization of the constants is fairly large. A more objective comparison is to display the mean logarithmic residual error versus the mean computational time, as in figure 3. The improvement provided by local optimization is still huge, typically three orders of magnitude at constant computation time. The syntactic restrictions also help to improve the convergence speed, but typically only by half an order of magnitude. Therefore the advantage of the typing system is partly that it improves GP convergence, but mainly that only valid solutions are generated. A conservative extrapolation of the computational time needed to reach an accuracy of $\log(d) = -4$ for the algorithm without local optimization leads to CPU time of at least one year. Let us emphasize that only this kind of qualitative improvement of the algorithm makes the problem of volatility forecast possible. Clearly, each algorithm should be used where it is efficient: the genetic program to explore the structure of the solution, and the local optimizer to adjust the tree's constants to the problem.

We have also done similar tests for $n = 3$, namely the fit of $\cos(x)$ on 3 periods. As this problem is mode difficult, the computational times grow, but the results are very similar. One noticeable difference is a long "latency" time at the beginning of the evolution, where the best solution is essentially the null function. This can be understood as the number of generations needed to discover a polynomial with a degree high enough (x^6) to replicate the oscillation of cosine over 3

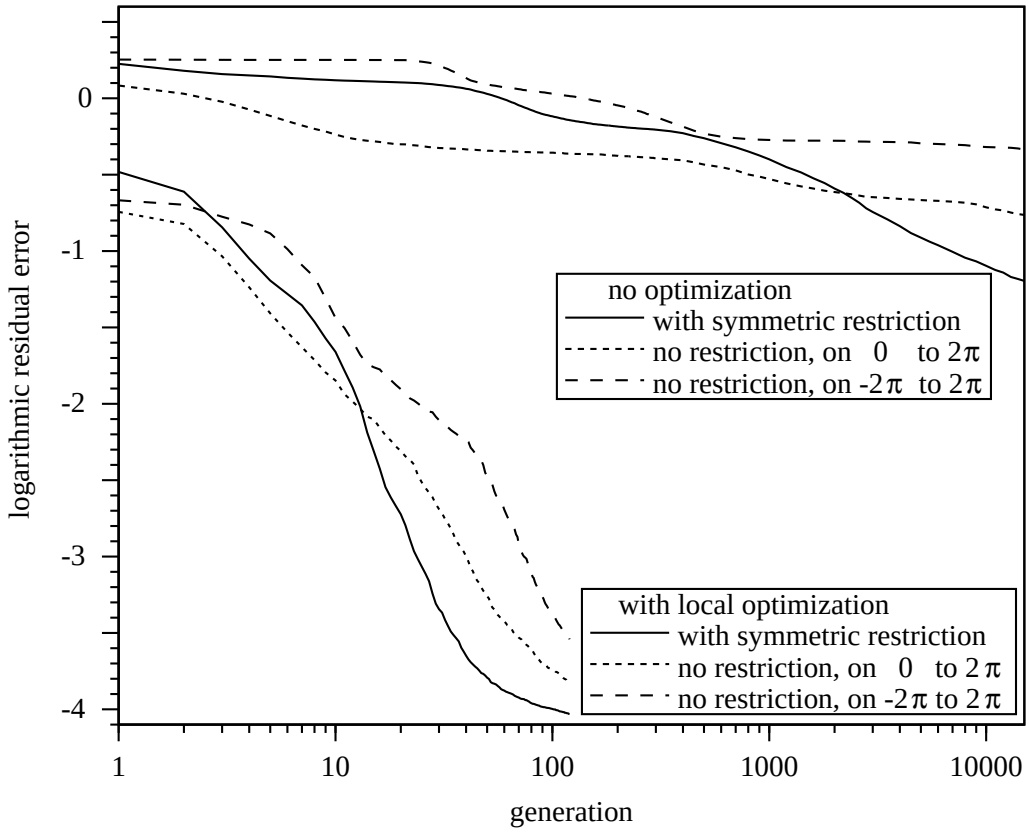

Figure 2: The mean logarithmic residual error versus the number of generations for different GP algorithms. The mean is computed over 200 GP runs, each with a population of 100 genes and an elitism rate of 50%. The runs are terminated after 120 generations with local optimization and after 15000 generations without local optimization.

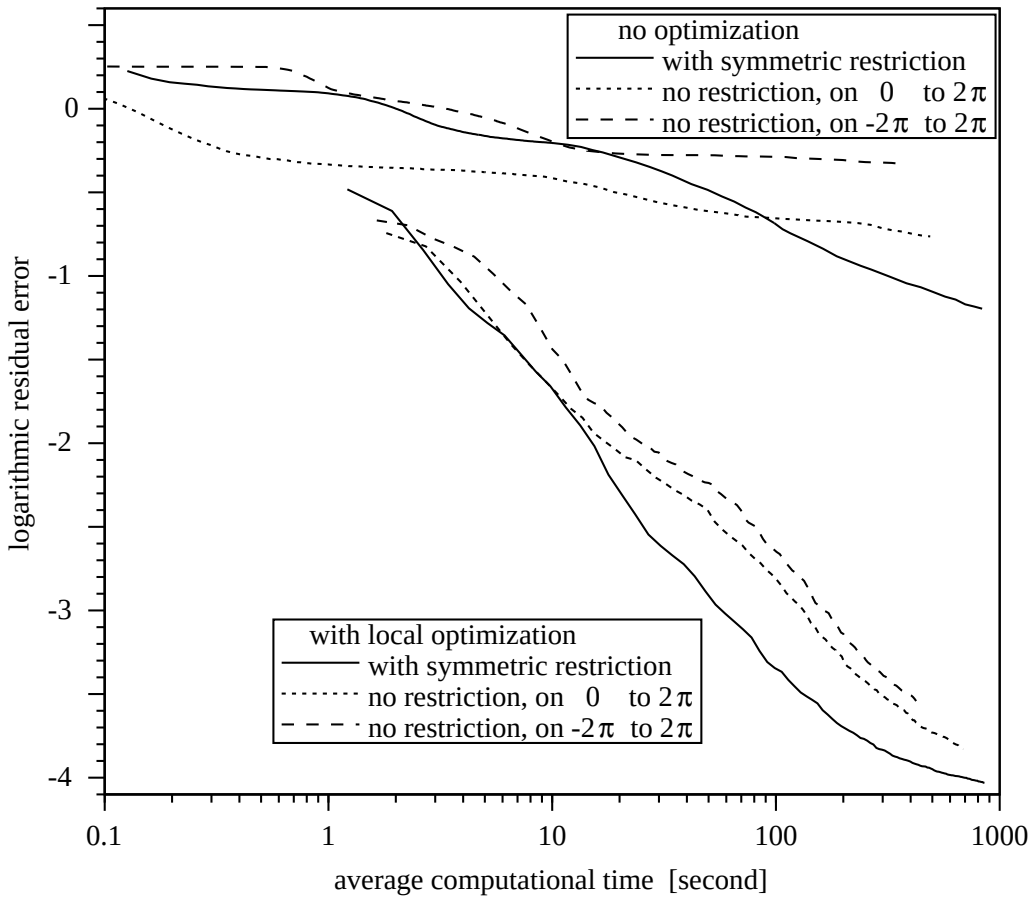

Figure 3: The mean logarithmic residual error versus the mean computational time for different GP algorithms. The GP parameters are as for fig. 2.

periods. After this latency time, the solutions improve rapidly.

Untill this point, we explored the statistical properties of GP algorithms. Yet, it is also interesting to look in more details at the specific forms of the solutions discovered by the GP. For the GP with syntactic restriction and local optimization, we performed a few runs over 200 generations with a population size of 100. Two examples of solutions discovered by the GP are

$$1 + -0.5 * x * (x + (x + (x + x * x * -0.018 * x) * x * -0.0164 * x) \\ * (x + x * x * -0.0168 * x) * -0.0833 * x)$$

and

$$(-0.405 + (x * (-0.196 + 0.0118 * x * x * (0.286 + -0.00252 * x^2)))^2) \\ * (-2.47 + x^2).$$

For both solutions, the distance to the $\cos(x)$ function (eq. 7) is of the order of $10^{-4.5}$, with maximal degree x^{10} and x^{12} respectively.

4 Volatility Forecasting Models Inference

4.1 Introduction

In the literature, there is a profusion of volatility forecasting models, but none of them is able to explain all the known empirical facts observed in high frequency foreign exchange time series. In this section, we use the strongly typed GP approach to explore the space of possible volatility forecasting models. With a GP approach allowing for a broad search in the space of the possible forecasts, we have good chances of finding models with new structures. Our goal is to discover if some specific structure appears in the best generated models and then to infer what is the class of the optimum volatility forecasting models to be used with foreign exchange rates.

In the case of foreign exchange rates, the basic time series is the middle logarithmic price $x(t) = [\ln(p_{\text{bid}}(t)) + \ln(p_{\text{ask}}(t))]$ / 2.0. Under the interchange of the per and exchanged currencies, the prices p are mapped into $1/p$, and the logarithmic prices into $x \to -x$. The main time series of interest is the annualized price changes (or returns) $r[\delta t]$ measured on a given time interval δt

$$r[\delta t](t) = \frac{x(t) - x(t - \delta t)}{\sqrt{\delta t / 1y}}. \tag{9}$$

The denominator annualizes the return, with $1y$ = 1 year. With this choice E[$r[\delta t]$] is essentially independent of δt, with a typical value around 10% for the main FX rates.

The volatility measures the fluctuation of the returns r, and must be identical for a specific rate or for the corresponding inverted rate. Accordingly, volatility models must be independent of the interchange of the per and exchanged currencies, namely they are required to be symmetric with respect to the change $r \to -r$

$$\sigma(-r) = \sigma(r). \tag{10}$$

Moreover, the volatility is always positive.

4.2 Daily Volatility Forecast

In this study, the quantity to be forecast is the *realized mean daily volatility* $\sigma_d(t)$, which is constructed from our time series of hourly returns $r[\delta t]$ as follows

$$\sigma_d^2(t) = \sigma^2[\Delta t, \delta t](t) = \frac{1}{24}\sum_{k=1}^{24} r^2[\delta t](t + k\delta t) \tag{11}$$

where δt is one hour and Δt is one day. Other values can be chosen for the parameters Δt and δt, but we have restricted our empirical study to the above values. For foreign exchange rates, statistical studies on observed price changes show no autocorrelation of the returns (except for very short time intervals), and no correlation between the return and the volatility. Therefore, we restricted our search to models which only predict the future volatility, but do not predict any future price change, namely we search for forecast of σ_d, and not for a joint forecast of σ_d and r.

The generated GP trees directly provide at each time t the forecast value $F^2_{\mathrm{GP}}(t)$ of the mean daily variance over the next 24 hours $\sigma_d^2(t)$, and the score function used to measure the quality of a forecast is the root-mean squared errors (RMSE)

$$\mathrm{score}^2 = E[\ (F_{\mathrm{GP}}(t) - \sigma_d(t))^2\ ]. \tag{12}$$

The fitness is computed from the logarithm of the score, with a penalty for complexity, as described in section 2.6. The volatility forecasts are built on the information contained in the historical data, namely the previous returns. As explained in the introduction, FX volatility models must be independent of the interchange of the per and exchanged currencies, leading to the exact symmetry $\sigma(-r) = \sigma(r)$. In GP, syntactic restrictions are used to impose this symmetry on the solution (see section 2.4). To enforce positivity, we have used a penalty approach in which all the GP trees which return a negative value for the variance are strongly penalized. In the selection mechanism for the generation of the next populations, these trees have a much lower probability for reproduction and then tend to disappear. We have used a function set restricted to simple arithmetic operations and operators commonly used in volatility process such as exponential moving averages (EMA), square and absolute functions. To test the impact of heterogeneous market components, we have introduced terminals corresponding to return measured on various time intervals.

4.3 Benchmark Models

The generated GP volatility models are compared to different theoretical volatility models which are in the framework of autoregressive conditional heteroskedastic (ARCH) models. These models are constructed from high frequency price changes, sampled at interval δt, and they are used to forecast mean volatility on a longer time interval Δt. In this study, we consider hourly returns δt = 1 hour, and forecast mean volatility for daily interval Δt = 1 day. The general form of the volatility models under consideration is

$$\begin{aligned} r(t+\delta t) &= \sigma_m(t+\delta t)\ \varepsilon(t+\delta t) \\ \sigma_m^2(t+\delta t) &= \sigma_m^2[\Omega(t), \theta] \end{aligned} \tag{13}$$

where ε is an unknown random i.i.d. (independent identically distributed) process, $\Omega(t)$ is the information set at time t which contains all previous return and volatility values, and θ are the process parameters. The time scale at which the process is evaluated is δt.

At a given time t, with the information set $\Omega(t)$, the volatility forecast for the volatility at $t+k\delta t$ is given by the conditional expectation

$$\widetilde{\sigma_m^2}[k\delta t](t) = E[\ \sigma_m^2(t+k\delta t) \mid \Omega(t)\ ]. \tag{14}$$

Using the process equations iteratively, the conditional expectation can be expressed as a function of the return and volatility contained in the information set $\Omega(t)$ at time t (see [Zumbach, 2000b]). At time t, the forecast for the average volatility on a time interval from t to $t+\Delta t$ is then given by

$$\overline{\widetilde{\sigma_m^2}}[\Delta t](t) = \frac{1}{n}\sum_{k=1}^{n} \widetilde{\sigma_m^2}[k\delta t](t) \tag{15}$$

where $\Delta t = n\delta t$. The parameters of the benchmark models are optimized by minimizing the root mean square error between the forecasted and realized average daily volatility

$$\mathrm{RMSE}^2 = E[\ (\sqrt{\overline{\widetilde{\sigma_m^2}}(t)} - \sigma_d(t))^2\ ]. \tag{16}$$

The first benchmark is the permanence hypothesis, namely the forecast is given by the historical volatility measured on the last day $\sigma_{\mathrm{hist}}[1\mathrm{d}]$ or the last week $\sigma_{\mathrm{hist}}[1\mathrm{w}]$. This model has no adjustable parameter. The second benchmark model we consider is the well known GARCH(1,1) model. The volatility process is given by

$$\sigma_m^2(t+\delta t) = \alpha_0 + \alpha_1 r^2(t) + \beta_1 \sigma_m^2(t). \tag{17}$$

To avoid recursion on the volatility term, this model can be rewritten using a moving average on the return, as follows

$$\begin{aligned} \sigma_m^2(t+\delta t) &= \sigma^2 + w(\sigma_1^2(t) - \sigma^2) \\ \sigma_1^2(t) &= \mu\sigma_1^2(t-\delta t) + (1-\mu) r^2(t) \end{aligned} \tag{18}$$

with three parameters σ, w and μ. For this model, the correlation decays exponentially, with a characteristic time of $\mu_{\mathrm{corr}} = \exp(-\delta t/\tau_{\mathrm{corr}})$ and $\mu_{\mathrm{corr}} = \alpha_1 + \beta_1 = \mu + w(1-\mu)$.

The third benchmark is the FIGARCH model [Baillie et al., 1996], which incorporate the long memory of the volatility through a fractional difference operator. The last type of benchmark model comes from the long memory Heterogeneous ARCH class of models. The HARCH model was developed by [Müller et al., 1997] and is based on many independent volatility components evaluated at different time horizons. The volatilities are computed from aggregated returns measured at time horizons ranging from 1 hour to a few weeks. The model can be formulated as follows

$$\begin{aligned} \sigma_m^2(t+\delta t) &= C_0 + \sum_{j=1}^{n} C_j \sigma_j^2(t) \\ \sigma_j^2(t) &= \mu_j \sigma_j^2(t-\delta t) + (1-\mu_j) r^2[k_j\delta t](t) \end{aligned} \tag{19}$$

where $C_j \geq 0$ for $j = 0 \ldots n$, and with the necessary stationarity condition $\sum_{j=1}^{n} C_j < 1$. The k_j are the aggregation factors of the returns and are chosen according to a geometric progression

$$k_j = p^{j-1}. \tag{20}$$

The same value $p = 4$ as in [Müller et al., 1997] is chosen here. The volatility memory of equation 20 is determined by the constant $\mu_j = \exp(-\delta t/\tau_j)$ with $\tau_j = \tau_0 k_j$. The present formulation differs in some details from the one used in [Müller et al., 1997], in particular we use annualized returns, leading to simpler equations.

4.4 Experiments on two FX rates

Inference of volatility forecasting models have been tested on two foreign exchange rates, i.e. USD-CHF and USD-JPY. For these experiments we have used 13 years of high frequency data from 1.1.1987 to 31.12.1999. In order to remove seasonality, hourly returns equally spaced in business time [Dacorogna et al., 1993, Breymann et al., 2000] are extracted from the high frequency data. The available data sets are divided in three subsamples. Data from 1.1.1987 to 31.12.1989 is used for the initialization of the moving average indicators, but not included in the score function. The in-sample data from 1.1.1990 to 31.12.1994 is used to compute the score function for the GP, and to optimize the parameters of the benchmarks models. The out-of-sample data from 1.1.1995 to 31.12.1999 is used for comparison of the performances of the various models. This is done to obtain models that capture the best structure of the volatility forecast, and not the particular data set used for the optimization. The out-of-sample data set contains 5 years of hourly data which represents more than 31,200 prices changes, or 1300 non-overlapping volatility observations.

We have experimented with various terminal sets and function sets. One terminal set is composed only of constants and hourly returns. In order to test the importance of aggregated returns, another terminal set is composed of constants and aggregated return at selected time intervals $k\delta t$ with k =1, 4, 24 (1 day), 120 (1 week) and 480 (1 month). The basic function set contains the addition, multiplication and EMA operator. In a slightly extended function set, we have added the the absolute and square value operators in order to generate models where σ^2 can contain terms in $|r|$.

The experiment reported below is done for both exchange rates using the four combinations of the two terminal sets and two function sets. A population of 100 individuals is evolved with a steady state genetic algorithm over 200 generations. At each generation, the best half of the population is kept, while the worst half is replaced (i.e. an elitism rate of 50%). The selection of the individuals for reproduction is done with the fitness proportional algorithm, as described in sec. 2.6. Optimization is always done with local optimization, and the vector of probabilities characterizing the Markov chain $\vec{p}_{\mathrm{MC}}$ is as given in sec. 3. In one case, the best model generated by the GP is not always positive in the out-of-sample, and we report the performance of the best GP tree with positive value in the out-of-sample.

The performance for the benchmark models and for the best GP trees are reported in the tables 1 and 2. For the HARCH model, the converge of the local search algorithm is problematic, and most of the time it converges to a point at the necessary stationarity condition $\sum_{j=1}^{n} C_j < 1$. For this model, the values reported in the table are the best results from 5 searchs with different starting

process	In-Sample	Out-Of-Sample	constants
$\sigma_{\text{hist}}[1\text{d}]$	5.00	4.30	0
$\sigma_{\text{hist}}[1\text{w}]$	4.38	3.97	0
GARCH(1,1)	4.20	3.84	3
FIGARCH	4.13	4.31	3
HARCH(6)	4.02	4.02	7
Run 1	3.98	3.76	6
Run 2	4.04	3.87	5
Run 3	4.00	3.74	9
Run 4	4.03	3.78	6

Table 1: Comparison of the RMSE values, in %, between the in-sample and the out-of-sample of the benchmark processes, and the best solution of each GP runs for the USD-CHF exchange rate.

points. Overall, the remarkable result is that the GP solutions are consistently better than the benchmarks, including out-of-sample. This clearly shows that GP, with the syntactic restrictions and the local optimization of the constants, is an efficient tool for discovering new forecasting models, without overfitting the data sets. The number of constants of the tree is as given by the best GP solution. Some of these constants can be redundant, and the number of independent constants might be smaller. Notice also that only the best tree for each run is reported. Yet, the next ranking genomes have also very good scores, and can be considerably simpler than the best individual. These solutions are indeed better at capturing the relevant structure of a good forecasting model.

A detailed examination of the best generated trees shows the following salient features:

- The EMA function is always used at least once in each tree, and most of the time an EMA operator is the root node. Clearly, a good forecast needs to have enough memory of the past, and this is achieved through the use of EMAs.

- The EMAs have mostly a constant range z, and not a variable range given by the value of a subtree. For example, subtrees of the kind $\text{EMA}[3.5;\ \cdots]$ are used, but subtrees of the kind $\text{EMA}[3.5+2.4*r^2;\ \cdots]$ seldomly appear. Moreover, the typical values for r^2 is the annualized variance, which for these data sets is of the order of 0.01 (i.e. $(10\%)^2$). For the last example, the changes in the EMA range are therefore numerically very small, and do not plays a role in the efficiency of this subtree. Only in a few cases, a numerically important variable EMA range appear in the final population.

- All the good solutions contain product of returns at different time horizons, for example like $r[\delta t]*r[24\delta t]$. Using the relation
$r[k\delta t](t) = r[m\delta t](t) + r[(k-m)\delta t](t-m\delta t)$, this term can be rewritten as $r[\delta t](t)*r[\delta t](t) + r[\delta t](t)*r[23\delta t](t-\delta t)$, namely as a square term, plus a *cross term of non overlapping returns*. When the terminal set contains only the return $r[\delta t]$ at the hourly time horizon δt

process	In-Sample	Out-Of-Sample	constants
$\sigma_{\text{hist}}[1\text{d}]$	4.51	6.19	0
$\sigma_{\text{hist}}[1\text{w}]$	3.98	5.96	0
GARCH(1,1)	3.81	5.62	3
FIGARCH	3.78	5.51	3
HARCH(6)	3.77	5.62	7
Run 1	3.74	5.40	2
Run 2	3.61	5.48	4
Run 3	3.60	5.44	5
Run 4	3.59	5.42	7

Table 2: Comparison of the RMSE values, in %, between the in-sample and the out-of-sample of the benchmark processes and the best solution of each GP runs for the USD-JPY exchange rate.

(but no aggregated returns), similar terms are generated through EMAs, like with $r[\delta t] * \text{EMA}[z; r[\delta t]]$. These cross terms are the key difference between the GP solutions and all the benchmark models that contain only quadratic terms. The reason for which these terms have been omited until now in volatility modeling is that $\text{E}[\ r[\delta t](t) * r[m\delta t](t - m\delta t)\ ] = 0$. Yet, the GP discovered that these terms are containing valuable informations on the evolution of the volatility.

5 Conclusion

In this study, we have used genetic programming with syntactic restrictions combined to local search techniques to explore the space of the volatility forecasting models for foreign exchange rates. Syntactic restrictions are used to impose a specific symmetry property on the solutions and local search is used to adjust the tree's constants to the problem. We have first tested this approach on the discovery of the transcendental function $\cos(x)$. This test shows that the improvement provided by local optimization of the tree's constants is huge, typically three orders of magnitude on such problems. The advantage of syntactic restrictions is partly that it improves GP convergence, but mainly that only solutions with the correct symmetry property are generated.

The application to the discovery of new types of volatility forecasting models for financial time series is a hard problem which is feasible only with local optimization of the constants. To reduce overfitting problems and to keep optimization time within reasonable bounds, a control of the GP trees' complexity is needed. We have used a penalization approach to promote trees of lower complexity with a reduced number of constant terminals.

For the volatility forecast problem, the experiment on two main exchange rates shows that the selected GP trees perform better than optimized benchmark processes. Moreover, the out-of-sample results indicate that these GP solutions are quite robust. The success of the best trees is

due to cross-product of two returns at different time horizons. Such cross terms are partly present in the HARCH process, but with fixed weights given by the coefficients C_j as this model is indeed quadratic in $r[k\delta t]$. In GP trees, the combination of the various cross terms is more complex and their optimum weighting can be directly evaluated. Let us notice that these cross terms are formed either by using moving average of microscopic returns or by directly including aggregated returns. The new cross products indicate qualitatively that the evolution of the volatility in the financial markets is sensitive to the past trend, where for example $r[\delta t](t) * r[m\delta t](t - m\delta t) > 0$, or to a sideway market, where for example $r[\delta t](t) * r[m\delta t](t - m\delta t) < 0$. This dependency of the realized volatility on the past price evolution is indeed a new stylized fact, whose discovery should be credited to GP. This new effect is clearly worth a detailed statistical analysis, which will be the subject of a forthcoming publication.

References

[Baillie et al., 1996] Baillie, R. T., Bollerslev, T., and Mikkelsen, H.-O. (1996). Fractionally integrated generalized autoregressive conditional heteroskedasticity. *Journal of Econometrics*, 74(1):3–30.

[Bhattacharyya et al., 1998] Bhattacharyya, S., Pictet, O. V., and Zumbach, G. (1998). Representational semantics for genetic programming based learning in high-frequency financial data. In Koza, J. R., Banzhaf, W., Chellapilla, K., Deb, K., Dorigo, M., Fogel, D. B., Garzon, M. H., Goldberg, D. E., Iba, H., and Riolo, R., editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 11–16, University of Wisconsin, Madison, Wisconsin, USA. Morgan Kaufmann.

[Breymann et al., 2000] Breymann, W., Zumbach, G., Dacorogna, M. M., and Müller, U. A. (2000). Dynamical deseasonalization in otc and localized exchange-traded markets. Internal document WAB.2000-01-31, Olsen & Associates, Seefeldstrasse 233, 8008 Zürich, Switzerland.

[Chopard et al., 2000] Chopard, B., Pictet, O. V., and Tomassini, M. (2000). Parallel and distributed evolutionary computation for financial applications. *Parallel Algorithms and Applications*, 15:15–36.

[Dacorogna et al., 1993] Dacorogna, M. M., Müller, U. A., Nagler, R. J., Olsen, R. B., and Pictet, O. V. (1993). A geographical model for the daily and weekly seasonal volatility in the foreign exchange market. *Journal of International Money and Finance*, 12(4):413–438.

[Koza, 1992] Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, USA.

[Montana, 1995] Montana, D. J. (1995). Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230.

[Müller et al., 1997] Müller, U. A., Dacorogna, M. M., Davé, R. D., Olsen, R. B., Pictet, O. V., and von Weizsäcker, J. E. (1997). Volatilities of different time resolutions – analyzing the dynamics of market components. *Journal of Empirical Finance*, 4(2-3):213–239.

[Press et al., 1986] Press, W. H., Flannery, B. P., Teukolsky, S. A., and Vetterling, W. T. (1986). *Numerical recipes. The art of scientific computing.* Cambridge University Press, Cambridge.

[Zumbach, 2000a] Zumbach, G. O. (2000a). The pitfalls in fitting garch processes. In Dunis, C., editor, *Advances in Quantitative Asset Management*. Kluver Academic Publisher.

[Zumbach, 2000b] Zumbach, G. O. (2000b). Volatility process and volatility forecast with long memory. Olsen & Associated internal paper.