A Genetic Programming Based Learning System to Derive Multipole and Local expansions for the Fast Multipole Method

Seyed Naser Razavi, Nicolas Gaud, Abderrafiâa Koukam, Naser Mozayani

Soft Computing and Multi-Agent Systems laboratory, Iran University of Science and Technology, Tehran, Iran

Multi-Agent Systems and Applications group, Laboratoire Systèmes et Transports, UTBM, 90010 Belfort, France. www.multiagent.fr

razavi@iust.ac.ir, nicolas.gaud@utbm.fr, abder.koukam@utbm.fr, mozayani@iust.ac.ir

Abstract

This paper introduces an automatic learning algorithm based on genetic programming to derive local and multipole expansions required by the Fast Multipole Method (FMM). FMM is a well-known approximation method widely used in the field of computational physics, which was first developed to approximately evaluate the product of particular $N \times N$ dense matrices with a vector in $O(N \log N)$ operations, while direct multiplication requires $O(N^2)$ operations. Soon after its invention, the FMM algorithm was applied successfully in many scientific fields such as simulation of physical systems (Electromagnetic, Stellar clusters, Turbulence), Computer Graphics and Vision (Light scattering) and Molecular dynamics. However, FMM relies on the analytical expansions of the underlying kernel function defining the interactions between particles, which are not obvious to derive. This is a major factor that severely limits the application of the FMM to many interesting problems. Thus, the proposed automatic technique in this article can be regarded as a very useful tool helping practitioners to apply FMM to their own problems. Here, we have implemented a prototype system and tested it on various types of kernels. The preliminary results are very promising, and so we hope that the proposed method can be applied successfully to other problems in different application domains.

Keywords - Fast Multipole Method, Genetic Programming, Local expansion, Multipole Expansion

1 Introduction

There are a large number of systems (physical, biological, etc.) that can be studied by simulating the interactions between the particles constituting the system. In many cases, the simulation of such systems requires evaluating all pairwise interactions between particles because each particle influences every other particle. Examples of such systems can be found in a wide variety of scientific domains, including: sociology, biology, physics, chemistry, ecology, economy, etc. The challenge of efficiently carrying out the related calculations is generally known as the *N-body* problem.

Since it is impossible to solve the equations of motion for a large ensemble of particles in closed form, *N*-body problems are solved using iterative methods. In an iterative method, the force on each particle is computed at each cycle, and this information is then used to update the state (i.e., the position and velocity) of each particle. Assuming *N* particles, a direct computation

of the forces requires $O(N^2)$ work per iteration. This $O(N^2)$ complexity severely limits the number of particles that can be simulated because of its rapid growth with N. In other words, the $O(N^2)$ complexity required by direct methods, makes large-scale simulations extremely expensive in some cases, and prohibitive in many other cases.

Several techniques have been proposed that may be used to reduce the complexity per iteration. Among these techniques, one can refer to the Fast Multipole Method (FMM) as one of the most successful ones. The FMM is an approximation algorithm originally proposed by Rokhlin as a fast scheme to accelerate the numerical solution of the Laplace equation in two dimensions [1]. It was further improved by Greengard and Rokhlin when applied to particle simulations [2, 3], and has since been identified as one of the ten most important algorithmic contributions in the 20^{th} century [4]. FMM can reduce the complexity of evaluating all pairwise interactions in large ensembles of N particles to $O(N \log N)$. This is a significant improvement over the $O(N^2)$ time required by direct methods, especially for very large values of N ($N > 10^6$).

Since its inception, FMM has been successfully applied to a wide variety of problems arising in diverse areas such as astrophysics, plasma physics, molecular dynamics, fluid dynamics, acoustics, electromagnetic, scattered data interpolation, and many more. Furthermore, It has found some applications in domains as seemingly unrelated as light scattering and radiosity calculations in computer graphics and vision [5, 6]. Recently, in [7, 8], the authors have introduced the potential use of the FMM in agent-based simulations, when there are a large number of interacting agents with complex interaction rules such as a physics-based flocking model.

However, the main problem with FMM is that its implementation relies on analytical expansions to approximate (in a suitable sense) the kernel function. That is, such expansions need to be carried out differently for different kernels. The kernel function is a function which defines the interaction laws between particles in the problem at hand. For a typical example of an interaction kernel, one can refer to the inverse square law such as Newton's gravitation law defining the interactions between bodies or Coulomb's law of electrostatics defining the interactions between charges. Even though many such approximations, often involving Legendre polynomials, Spherical Harmonics and Bessel functions, have been derived for many applications, many users find it very difficult or cumbersome to derive new expansions for new kernels, assuming such expansions can be found analytically.

So far, a few methods have been developed to deal with the above problem [9-13]. These methods are generally known as *kernel-independent* fast multipole methods in the sense that they do not rely on any analytic expansions and utilize only kernel evaluations. Unfortunately, these methods have not received enough attention, despite their scientific and technological contributions.

Based on our previous experiences in implementing and working with such methods as described in [7, 8], we believe that one explanation for this could be the complexity that these methods introduce to the FMM. FMM on its own is a very complicated algorithm, and these methods make the situation even worse. However, the most important reason could be related to the fact that these methods are usually less accurate compared to the FMM, and at the same time they are computationally more expensive. These undesired features significantly decrease their chances to be used in scientific computation and real-time applications, which are the main target domains for the original FMM. Additionally, these methods usually make some limiting assumptions about the kernel which are invalid for many kinds of kernels.

This article introduces a new GP-based automatic learning technique, which can be used to derive different expansions required in the FMM. Contrary to the kernel independent methods, this approach does not have any negative impact on the efficiency and accuracy of the FMM (except the time required to run the GP system to find analytical expansions needed by FMM, which is completely negligible to the time required by FMM to simulate a large ensemble of particles). Several experiments performed on different kernels confirm that the GP system can be used to evolve exact analytic expansions of the kernel which can be served to construct an accurate and efficient implementation of the FMM algorithm, if a sufficient amount of time is provided to the system. Moreover, the GP system can be used as a "black box" method which is applicable to arbitrary kernels. Therefore, applying the method should then simply be a matter of installing a library and providing a user-defined routine to evaluate the kernel at a given point. Thus, in contrast to the kernel independent methods, the complexity of our proposed technique is completely hidden from the end-user.

The rest of this paper is organized as following: Section 2 defines the problem of finding analytical expansions for a given kernel in more detail. Section 3 describes the GP system which is used to solve our target problem in this article. Some experimental results are discussed in Section 4. Finally, a summary of this work along with some future research guidelines is provided in Section 5.

2 FMM and kernel expansions

Let us assume that there are N source densities u_i located at x_i $\{1 \leq i \leq N\}$ in a d-dimensional space (d = 2 or 3). All we need is to compute the potential v_j at M target points y_j $\{1 < j \leq M\}$ induced by a kernel K using the following summation:

$$v_j = \sum_{i=1}^{N} K(x_i, y_j) u_i = \sum_{i=1}^{N} K_{ij} u_i, \quad j = 1, \dots, M \tag{1}$$

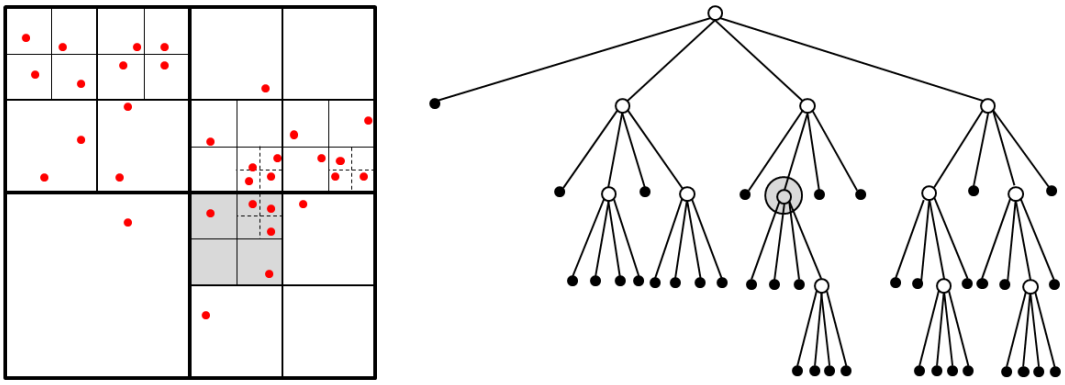

Figure 1 A 2D particle distribution (left) and its corresponding quadtree (right).

Clearly, a direct implementation of the above summation requires O(MN) operations to compute all pairwise interactions between source points and target points. In many applications, the set of source points and targets points are identical (each point induces some potential on other points, and at the same time it is influenced by other points). In such cases, we have $M = N$, and the complexity is thus equal to O(N^2), which is obviously prohibitive for large values of N. The FMM algorithm can reduce the complexity of the above computations from O(N^2) to O(N log N), which is a significant reduction specially for a very large N ($N \geq 10^6$).

FMM achieves its performance by introducing a hierarchical partition of a bounding square D, enclosing all particles, and two series expansions for each box at each level of the hierarchy. More precisely, the root of the tree is associated with the square D and referred to as level 0. The boxes (squares) at the level $l + 1$ are obtained recursively, subdividing each box at level l into four squares, referred to as its children. The tree is constructed so that the leaves contain no more than a certain fixed number of particles, say s. For non-uniform distributions, this leads to a potentially unbalanced tree, as shown in Figure 1 (which assumes $s = 1$). This tree is the main data structure used by the FMM.

The idea behind the space partitioning is to group source points into clusters (boxes in 2D space and cubes in 3D space) and consider the whole cluster as one point which approximates the influence of the source points to well-separated targets. The same idea can be applied to target points. That is, when the target points are far enough from the source points, targets can also be grouped into clusters. This way, it is possible to evaluate the contribution of the source points inside A to target points inside B at a single step, reducing the amount of computational efforts needed, assuming that A and B are two well-separated clusters. The situation is shown in Figure 2.

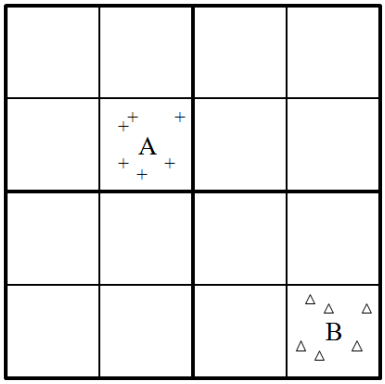

Figure 2 *A* and *B* are two well-separated clusters. Cluster *A* contains some source points marked with "+" and cluster *B* contains some target points marked with "Δ". Instead of computing the pairwise interaction between each source from *A* and each target from *B*, The FMM algorithm computes the potentials of target points inside *B* due to source points inside *A* in a single computational step.

The above idea in FMM is implemented using expansion operations. In fact, two types of expansions are used in the FMM: the *multipole expansion* and the *local expansion*. The multipole expansion for a box *B* encodes the contribution of *B* due to the source densities inside it to the far-field (non-adjacent boxes). Inversely, the local expansion for *B* encodes the contribution from the far-field to the target points inside *B*. For a box *B*, the multipole expansion depends only on the source points inside it, and hence it can be computed only once and then can be reused for any target box in the far-field. Similarly, the local expansion for box *B* depends only on the targets inside it, and again, it can be computed only once and reused for any source box in the far-field. This way, FMM can save a large amount of computations.

2.1 MULTIPOLE AND LOCAL EXPANSIONS

The implementation of the FMM relies on the analytic expansions (both multipole and local expansion) of the underlying kernel function. That is, if the kernel $K(x_i, y_j)$ is separable then it can be factorized as

$$K(x_i, y_j) = \sum_{m=0}^{\infty} a_m(x_i, x_*) f_m(y_j, x_*) \cong \sum_{m=0}^{p-1} a_m(x_i, x_*) f_m(y_j, x_*) \tag{2}$$

where x_* is any point other than x_i in the plane and represents the center of expansion. Note that in the above factorization, the first function $a_m(x_i, x_*)$ depends only on variable x_i (source points) and the second function $f_m(y_j, x_*)$ depends only on variable y_j (target points). These two functions depend on the kernel function, and hence vary from one kernel to another. As an example, please refer to Section 2.1.1 and Section 2.1.2 to see the factorization for kernel function $K(x_i, y_j) = \log\|y_j - x_i\|$ and the corresponding functions $a_m(x_i, x_*)$ and $f_m(y_j, x_*)$.

Now the potential v_j, defined in (1), can be evaluated in the following way:

$$
\begin{aligned}
v_j &= \sum_{i=1}^{N} K(x_i, y_j) u_i \\
&\cong \sum_{i=1}^{N} \sum_{m=0}^{p-1} a_m(x_i - x_*) f_m(y_j - x_*) u_i \\
&= \sum_{m=0}^{p-1} \sum_{i=1}^{N} a_m(x_i - x_*) f_m(y_j - x_*) u_i \\
&= \sum_{m=0}^{p-1} f_m(y_j - x_*) \sum_{i=1}^{N} a_m(x_i - x_*) u_i \\
&= \sum_{m=0}^{p-1} c_m f_m(y_j - x_*)
\end{aligned}
$$

Where $c_m = \sum_{i=1}^{N} a_m(x_i - x_*) u_i$.

Clearly, c_m is only dependent on the source points and thus it can be computed for a group of source points only once and can be reused for several different target points.

2.1.1 Multipole expansion

It is more convenient to describe FMM using a simple example kernel. Here, we use the simple kernel $K(x_i, y_j) = \log\|y_j - x_i\|$ for this purpose. As mentioned earlier, the main idea of FMM is to represent the potentials of a set of source densities using multipole and local expansions at places far away from these sources. Let's assume that *n* source densities are located inside a disk centered at x_* with radius *r*, as shown in Figure 3. Then for every point *y* outside the disk with radius *R* (*R* > *r*), the potential v_y at *y* due to the source densities inside the smaller disk can be represented by a set of coefficients $c_m (0 \le m < p)$, where

$$
v_y = c_0 \log(y - x_*) + \sum_{m=1}^{p-1} \frac{c_m}{(y - x_*)^m} + 0\left(\frac{r^p}{R^p}\right) \tag{3}
$$

in which $0\left(\frac{r^p}{R^p}\right)$ is a residual term and c_m satisfies:

$$
c_0 = \sum_{i=1}^{n} u_i, \quad c_m = \sum_{i=1}^{n} \frac{-(x_i - x_*)^m}{m} \cdot u_i
$$

The expansion defined by (3) is called *multipole expansion*. Comparing this result with the equation defined in (2), gives the following factorization of the kernel $K(x_i, y_j) = \log\|y_j - x_i\|$:

$$
a_m(x_i, x_*) = \begin{cases} 1, & m = 0 \\ -\dfrac{(x_i - x_*)^m}{m}, & m \ge 1 \end{cases} \tag{4}
$$

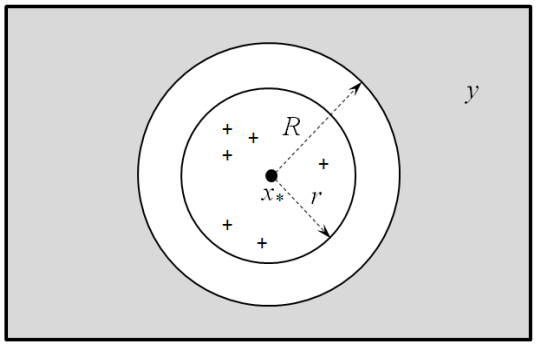

Figure 3 Multipole expansion at center x_* which is valid only outside the bigger disk (the gray region in the figure)

and

$$f_m(y_j, x_*) = \begin{cases} \log(y_j - x_*) & m = 0 \\ \dfrac{1}{(y_j - x_*)^m}, & m \geq 1 \end{cases} \tag{5}$$

In this article, our goal is to develop a system to derive these two functions which can be used to construct the multipole expansion (or local expansion) required by the FMM method.

2.1.2 Local expansion

On the other side, if the source densities are all located outside the disk with radius *R*, then the potential v_y at any point *y* inside the disk with radius *r* can be represented with a set of coefficients $c_m (0 \leq m < p)$, where

$$v_y = \sum_{m=0}^{p-1} c_m \cdot (y - x_*)^m + O\left(\frac{r^p}{R^p}\right) \tag{6}$$

with c_m satisfying:

$$c_0 = \sum_{i=1}^{n} u_i \log(x_* - x_i), \quad c_m = \sum_{i=1}^{n} \frac{-1}{m \cdot (x_i - x_*)^m} \cdot u_i$$

This is called *local expansion*. In both expansions, the truncation number *p* is usually a small constant determining from the desired accuracy of the result. A larger value for parameter *p* generally results in more computational times and at the same time increases the accuracy of computations.

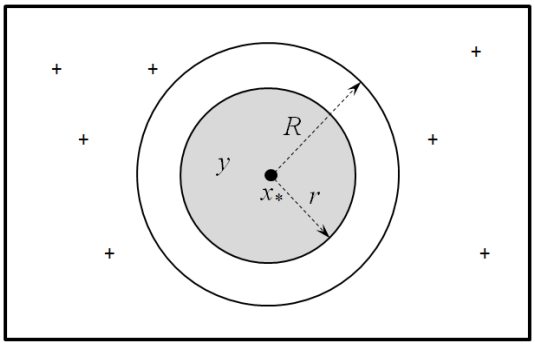

Figure 4 Local expansion at center x_* which is valid only inside the smaller disk (the gray region in the figure).

2.2 THE FMM ALGORITHM

After partitioning the space into clusters and constructing the hierarchical tree structure, in which every node corresponds to a geometric box in the computational domain, FMM performs two passes on the tree: the *upward pass* and the *downward pass*. The upward pass is a bottom-up traversal of the tree in which a *p*-term multipole expansion is formed at every node of the tree. At the finest level, the multipole expansions are computed directly, while the multipole expansions of internal nodes at higher levels of the tree are formed by shifting the multipole expansions of the child nodes to the center of their parents and adding them together.

Having the multipole expansions at every node, a top-down traversal of the tree starts to compute the local expansions at every node. The local expansion at a child node is constructed by shifting the local expansion at the parent to the child's center, shifting the multipole expansions of well-separated children of the nearest neighbors of the parent of the node to its center and adding them together. Finally, the local expansions at every leaf node are evaluated to compute the contribution from far-field to the particles inside that node. This far-field contribution it then added to the near-field interactions computed by iterating over all the source points in the neighborhood of the target box to obtain the potential of each target point. For a more detailed description of the FMM algorithm, see [14].

Next section describes a GP system that can be used to automatically derive the two functions $a_m(x, x_*)$ and $f_m(y, x_*)$ for both factorizations in multipole expansion and local expansion of any arbitrary kernel.

3 Genetic Programming

Evolutionary computation, as the name suggests, is a kind of computation inspired from the process of natural evolution. It involves a family of algorithms called Evolutionary Algorithms (EA). Each algorithm in the family implements the same idea of genetic search in a different way. One of the main differences between the members is the data structure (chromosome) they use to encode a candidate solution. They can use simple structures like binary strings or more complex structures such as trees or graphs.

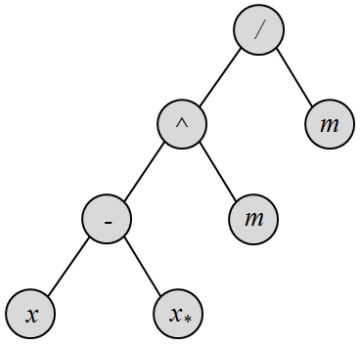

Figure 5 a tree structure for the model: $(x - x_*)^m/m$

Genetic Programming (GP), first introduced by Koza [15], uses tree structures (e.g. syntax tree) to represent solutions to a given problem. So GP can be viewed as a good candidate whenever candidate solutions to a problem can be naturally represented by trees. This representation is extremely flexible, since trees can represent computer programs, mathematical equations or complete models of process systems. In this application, our goal is to find formulas which best approximate multipole or local expansions in the FMM method and so it seems rational to use GP for this application. Another advantage of using GP for this problem is that unlike other methods such as neural networks, the solution found by GP (the expression trees representing analytical expansions of the underlying kernel) is easily readable and comprehensible for human. Furthermore, unlike some other function approximation approaches, the user can apply GP to its own problem without the need to be familiar with computer programming or advanced mathematical techniques. This becomes more important if we consider that the users of the FMM come from many different and diverse areas like those mentioned in the introduction section.

Like any other evolutionary algorithm, GP works with a set of individuals which together form a population of candidate solutions. At each cycle, the algorithm evaluates the individuals of the current population, selects better ones for reproduction, generates new individuals by performing genetic operators such as crossover and mutation operators, and finally creates the new population by replacing some of the old individuals with the new ones. The new generation goes through the same process to create another generation. This process is repeated until a termination criterion (such as finding a solution with required quality, spending a specified amount of time, or a combination of both) is satisfied. The fittest individual in the process serves as the final solution.

3.1 MODEL REPRESENTATION IN GP

In contrast to the common optimization methods, in which potential solutions are represented as numbers, in GP the potential solutions are usually represented by a nonlinear structure consisting of several symbols. Tree structures are one of the most popular methods for representing candidate solutions because of their flexibility to represent computer programs, mathematical equations, logical formulas and many others.

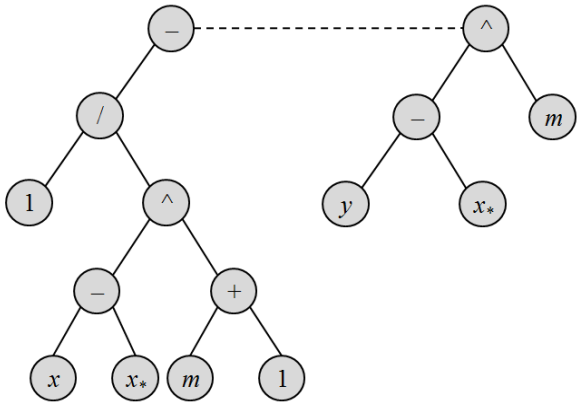

Figure 6 An individual representing the factorization in (7)

The first step in designing a GP system is to decide about two important sets used to construct the tree structures: *Terminal set* (*T*) and *Function set* (*F*). For example, the set of operators *F* can contain the basic arithmetic operations (+, -, ×, /) as well as other mathematical functions, Boolean operators (and, or, not, etc.), conditional operators or any user-defined operators. The set of terminals *T* provides the required arguments for the functions in *F*. A typical example for the terminal set is $T = \{x, y, \mathbb{R}\}$ with *x* and *y* being two independent variables, and $\mathbb{R}$ represent the set of real numbers. Therefore, a candidate solution (program) may be depicted as a rooted, labeled tree using functions (internal nodes of the tree) from the function set *F* and arguments (leaf nodes of the tree) from the terminal set *T*.

In this work, we wish to find a factorization of a given kernel $K(x, y)$ representing multipole expansion or local expansion for that kernel (see Section 2.1). Therefore, in our GP system, each individual consists of at least two tree structures, one representing $a_m(x, x_*)$ and the other one representing $f_m(y, x_*)$. The number of trees in each individual may be more than two depending on the given kernel function. That is, in the factorization of a given kernel, the first term in the *p*-term expansion may differ from the other terms. For more detail on this, please see Section 4.1. In our implementation, both types of individuals are allowed to exist in the same population. Figure 6 shows an example individual including two trees representing the factorization given in (7).

$$\frac{1}{y-x} = \sum_{m=0}^{\infty} -\frac{1}{(x-x_*)^{m+1}} \cdot (y-x_*)^m \tag{7}$$

3.2 INITIALIZATION

The initial step in GP is the creation of an initial population. Generally, at this step, a pre-specified number of individuals are randomly created in order to achieve a high degree of diversity. In GP, there are two common methods to create the initial population: *grow* and *full*. In the grow method, different branches in a tree can have different lengths, while in the full method all branches should have the same length (i.e., all leaf nodes should be at the same depth). In

order to achieve a better diversity, Koza suggests a third method called *ramped half-and-half* [15]. In this method, half of the population is created using the *grow* method, and the other half is created using the *full* method. The trees are constructed using different heights ranging from zero to the maximum initial height specified by the designer. This is the method of choice in many applications because of its ability to create a very diverse population. For the same reason, this method has been used in our implementation.

3.3 FITNESS FUNCTION AND SELECTION

After the creation of initial population, each individual in the population should be evaluated to prepare the population for the selection phase. Individuals are evaluated using a function which is called *fitness function*. This function assigns a real number to each individual indicating its quality or goodness in solving the problem at hand. Selection is then performed based on the individuals' fitness so that a better individual is more likely to be selected. This follows the "survival of the fittest" principle which occurs in the nature.

In a symbolic regression problem, the fitness function is usually based on the square error between the estimated and desired output. In this work, the same approach is used to evaluate potential solutions in the GP system. The first step in evaluating a given kernel is to create some number of fitness cases, let's say *n*. This number is an input parameter of the GP system and is determined by the designer. In this particular example, each fitness case is an ordered pair in the form of $\big((x_i, y_i), K(x_i, y_i)\big)$ with x_i and y_i being two randomly selected points in the complex plane satisfying the conditions defined in Section 2.1, and $K(x_i, y_i)$ is the value of the kernel function at (x_i, y_i). To evaluate the fitness of an individual, it is applied to every fitness case like (x_i, y_i) and its value $\widehat{K}(x_i, y_i)$ is recorded. Then, the error related to the *i*th fitness case can be computed as following:

$$error_i = K(x_i, y_i) - \widehat{K}(x_i, y_i) \tag{8}$$

Finally, the total error for an individual is computed by summing the square errors over all of fitness cases as defined in the following equation

$$total\ error = \sum_{i=1}^{n} error_i^2 \tag{9}$$

which obviously should be minimized.

An alternative to the well-known *sum of squares* function is to use the *number of hits* as a measure of fitness. To use this measure, first we establish the required precision for hitting $K(x_i, y_i)$ by $K(x_i, y_i)$, say ϵ. In other words, a hit occurs whenever $|K(x_i, y_i) - K(x_i, y_i)|$ is less than or equal to ϵ, in which ϵ is a very small positive pre-specified by the designer. Then, we can measure the fitness of each individual by simply counting the number of hits. Later in Section 4, we will use both of these measures to evaluate the quality of the solutions found by GP.

After evaluating a population, in the selection step, the algorithm selects the parents of the next generation and determines surviving individuals from the current generation. *Tournament* selection is the most widely used selection strategy in GP, and we have used this strategy in our implementation. In tournament selection, to select a parent from the current population, k individuals are selected at random from the population, and the fittest one (winner) is selected to be a parent. The parameter k is called *tournament size* and should be specified by the designer of the GP system. Several reasons are reported in the literatures for the popularity of tournament selection. One reason is its simplicity. Another important reason is that tournament selection does not rely on the knowledge of the entire population. This becomes more significant when, for example, the population size is amazingly large, or the population is distributed in some way (perhaps on a parallel system) [16].

3.4 GENETIC OPERATORS

After selecting parents from the current population, the algorithm generates new candidate solutions by applying genetic operators on the parents. The most widely used genetic operators in GP are *crossover*, *mutation* and *direct reproduction*. Crossover is a binary operator which takes as input two individuals and produces two offsprings by exchanging random parts of the two parents. In GP, a frequently used crossover is the *subtree crossover* which exchanges two randomly selected subtrees in the parents to produce two new individuals (see Figure 7). As individuals in our GP system may consist of several trees, it is worth mentioning that in our implementation, each tree in a given individual is recombined with its corresponding tree in the other individual. In mutation, a small random change is performed on the parent to produce one new individual. There are several mutations specially designed for the tree structures in GP such as point mutation, subtree mutation (see Figure 8), shrink mutation and hoist mutation. These mutations are examples of fair-size mutation, as they try to avoid producing very big offsprings during mutation. The reason for the existence of these operators is to control a common undesired phenomenon called *bloat*. This phenomenon along various methods to control it is discussed in more detail in Section 3.6. For a good introduction on this subject, the reader may refer to [17]. Based on the suggestions provided in [18], a combination of these mutations is used in our implementation. In fact, whenever a mutation is needed to be performed, one of the above mutation operators is selected at random and then applied to the selected parent. Finally, the reproduction operator simply puts the selected parent directly into the new population without any change.

Unlike GA in which the crossover and mutation are performed sequentially, in GP only one of these operators is selected at random and performed on each parent to generate new individuals. The probability of performing crossover on a selected individual is p_c, the probability of mutation is p_m and the probability of direct reproduction is $1 - p_c - p_m$.

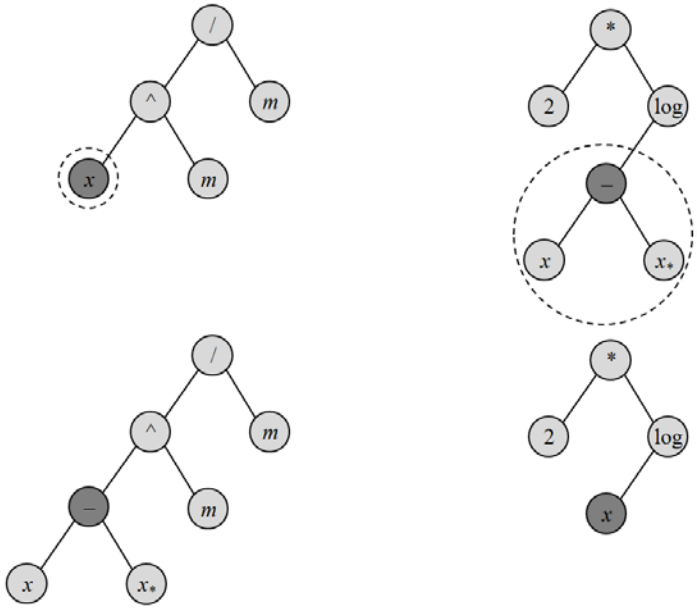

Figure 7 An example for subtree crossover with crossover points highlighted. The top part of the figure shows two parents and their randomly selected subtrees. The two trees in the bottom part are the resulting offsprings produced by exchanging subtrees in their parents.

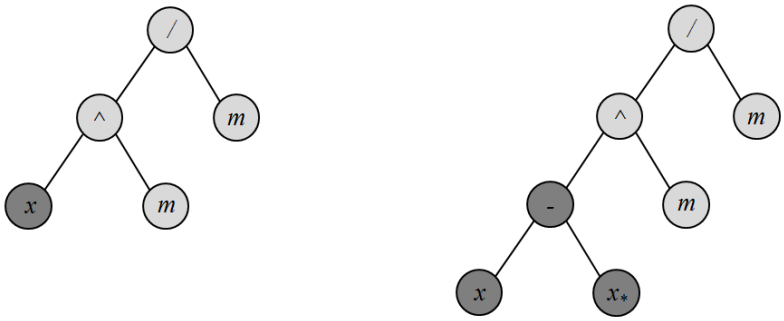

Figure 8 An example for subtree mutation with mutation point highlighted. The left part shows a parent and a randomly selected node of it. The right part shows the new offspring along with the new subtree which is located in the place of removed subtree in the parent.

3.5 REPLACEMENT STRATEGY

As the population size is fixed, not all the new individuals and the old ones can survive. Therefore, before inserting new individuals into the population, a mechanism is needed to remove some of the old individuals to make room for the newcomers. This mechanism is called *replacement strategy*, and it is performed at the end of each evolution cycle. We have used *elitist* replacement strategy in order to keep the best individuals of the old population with a *generational gap* parameter P_{gap}. For example, a value of 0.9 for this parameter means that 90% of the old population is killed and only the best 10% will survive. A very low value for this parameter increases the risk of *premature convergence* (getting stuck in a local optimum at the early stages of the GP run).

3.6 BLOAT

Bloat can be defined as uncontrolled growth of GP individuals without any remarkable increase in their fitness [17]. It has several significant practical effects: large individuals are computationally expensive to evolve and use, it is not easy to interpret them and furthermore, they may exhibit poor generalization. For these reasons, any GP system needs some mechanisms to keep away from it. Bloat has been a subject of intense study since the early years of GP inception and still it is an ongoing research area. Because of its practical effects on a GP system, a variety of effective techniques have been proposed to control bloat [19, 20]. This section describes the technique used in our proposed system to control bloat.

A simple yet effective way to control bloat is using genetic operators which directly or indirectly have anti-bloat effect. For example, one can refer to *size-fair crossover* and *size-fair mutation* among the most recent methods [21, 22]. In size-fair crossover, the first subtree in the first parent is selected randomly and its size is computed. The size of this subtree is then used to constrain the choice of the second subtree in the second parent to ensure that the second subtree will not be "unfairly" big. For mutation, several methods have been proposed to counteract bloat. For example, in *hoist mutation*, the new subtree is selected from the subtree being removed from the parent to guarantee that the new offspring will be smaller than its parent [23]. *Shrink mutation* is a special case of subtree mutation where the randomly chosen subtree is replaced by a randomly chosen terminal [24]. For a good introduction to the mutation operators, the reader may refer to [17]. In [18], the authors argue that combinations of subtree crossover and subtree mutation operators can control bloat in linear GP systems.

In addition to anti-bloat genetic operators discussed above, there are some anti-bloat selections strategies that may be used to control bloat in GP systems [15, 25-28]. In this work we have followed the approach proposed in [27]. In this method, known as *parsimony pressure*, the fitness function is defined to be $f(i) - c \times l(i)$, where $f(i)$ is the raw fitness of individual i defined by (8) and (9), $l(i)$ is its size, and c is a constant known as *parsimony coefficient*. In our implementation, this coefficient is adaptively adjusted at each generation using the following formula:

$$c = \frac{Cov(f, l)}{Var(l)}$$

In this formula, Cov(f, l) and Var(l) represent covariance and variance respectively. However, most implementations actually keep the parsimony coefficient constant.

3.7 DESIGN SUMMARY

Table 1 summarizes the designing parameters of the GP system used to derive multipole and local expansions for various kernels. Also, the values for the most important parameters are presented in Table 2, which are obtained mainly by try and error. The reason is that the optimal

values for these parameters depend too much on the details of the particular problem at hand, and hence it is impossible to make general recommendations for setting optimal values. However, the good news is that GP is very robust in practice, meaning that it is likely that many different parameter settings will work. There are several suggestions and rules, which may be useful in some situations, but the best values for these parameters are often determined by trial and error. However, as a consequence of GP robustness, one need not typically spend a long time tuning GP for it to work adequately.

Table 1 basic components of our GP system

Objective	Finding the multipole (or local) expansion for a given kernel function
Terminals for $a_m(x, x_*)$	Complex variables x, x_*; integer variable m; and random constants chosen from [-5, 5]
Terminals for $f_m(y, x_*)$	Complex variables y, x_*; integer variable m; and random constants chosen from [-5, 5]
Function set	+, -, ×, /, exp, pow, log, factorial; operating on complex numbers as well as real numbers
Fitness function	Sum of the squared errors over 100 fitness cases as defined in (9)
Selection	Tournament selection with tournament size 50
Initialization	Ramped half-and-half (depth 0 to 4)
Crossover	Size-fair crossover
Mutation	A combination of size-fair mutations including: point mutation, shrink mutation and hoist mutation
Parameters	See Table 2
Termination	Finding a solution with a total error less than 10^{-5} or reaching to the maximum number of generations
Bloat	Anti-bloat selection and anti-bloat genetic operators

Table 2 Parameters of the GP system and their values

Parameter	**notation**	**value**
Population size	μ	500, 1000, 2000, 5000
Tournament size	k	$0.05 \times \mu$
Number of fitness cases	n	100
Crossover rate	p_c	0.90
Mutation rate	p_m	0.01
Generational gap	P_{gap}	0.90
Maximum initial depth	id_{max}	4
Maximum depth	d_{max}	10
Maximum number of generations	g_{max}	50
Maximum size of trees	s_{max}	200

4 Experimental results

This section presents the results of the experiments which are performed to show the practical effectiveness of the proposed GP system. All the results reported in this section are obtained by averaging over 20 runs. Also, All experiments were conducted on a desktop computer configured

with one 2.5GHz Quad-Core Intel Pentium processors and 4GB RAM running a Windows 7 Professional x32 Edition. Our application was developed in Java and the Java VM used to execute the tests was configured with 1GB memory.

Table 3 and Table 4 introduce three different kernels which are used in the experiments. We have used these kernels because they are frequently used in the literature and also their expansions are known. This will enable us to compare the results of the GP system to the known solutions. For each kernel, the corresponding factorization for multipole expansion and local expansion is also provided for comparison reasons. Table 5 presents sample solutions found by GP system related to the multipole expansions of the given kernels. By simplifying these solutions using simple mathematical operations, it is easy to verify that the given solutions in Table 5 are exactly equal to their corresponding multipole expansion given in Table 3. However, one does not necessarily need the simplified versions of the solutions obtained by GP to be able to evaluate them. Like other machine learning tasks, the process can be done automatically by applying the solutions to some test sets and to see how well a particular solution can generalize for new data. Here, as the solutions found by the GP system are exactly equal to the optimal ones, we have disregarded the testing phase for this particular case.

Table 3 Three different kernels used in the experiments and their multipole expansions

Kernel	$K(x,y)$	$a_m(x,x_*)$	$f_m(y,y_*)$
I	$\frac{1}{y-x}$	$(x-x_*)^m$	$\frac{1}{(y-x_*)^{m+1}}$
II	$\log(y-x)$	$\begin{cases}1, & m=0\\ -\frac{(x-x_*)^m}{m}, & m\geq 1\end{cases}$	$\begin{cases}\log(y-x_*), & m=0\\ \frac{1}{(y-x_*)^m}, & m\geq 1\end{cases}$
III	$e^{-(y-x)^2}$	$e^{-(x-x_*)^2}\sqrt{\frac{2^m}{m!}}(x-x_*)^m$	$e^{-(y-x_*)^2}\sqrt{\frac{2^m}{m!}}(y-x_*)^m$

Table 4 different kernels used in the experiments and their local expansions

Kernel	$K(x,y)$	$a_m(x,x_*)$	$f_m(y,y_*)$
I	$\frac{1}{y-x}$	$\frac{-1}{(x-x_*)^{m+1}}$	$(y-x_*)^m$
II	$\log(y-x)$	$\begin{cases}\log(x_*-x), & m=0\\ -\frac{1}{m(x-x_*)^m}, & m\geq 1\end{cases}$	$(y-x_*)^m$
III	$e^{-(y-x)^2}$	$e^{-(x-x_*)^2}\sqrt{\frac{2^m}{m!}}(x-x_*)^m$	$e^{-(y-x_*)^2}\sqrt{\frac{2^m}{m!}}(y-x_*)^m$

Table 5 Sample solutions found by GP system for each kernel (multipole expansions)

Kernel I	$a_m(x, x_*)$ = POW (SUB (x, x_*), MUL(DIV (1.0, 1.0), m)) $= (x - x_*)^m$
	$f_m(y, x_*)$ = DIV (POW (SUB (y, x_*), DIV (m, -1.0)), (SUB (y, x_*)) $= 1/(y - x_*)^{m+1}$
Kernel II	$a_0(x, x_*)$ = 1.0 $a_m(x, x_*)$ = DIV (DIV (POW (SUB (x, x_*), m), POW (ADD (0.0, POW (m, 0.0)), LOG (POW (1.0, DIV (m, LOG (LOG (m))))))), m)
	$f_0(y, x_*)$ = MUL (LOG (SUB (y, x_*)), POW (m, SUB (2.0, 2.0))) $f_m(y, x_*)$ = DIV (DIV (POW (1.0, m), POW (SUB (y, x_*), m)), DIV(m, m))
Kernel III	$a_m(x, x_*)$ = MUL (POW (SUB (x, x_*), m), MUL (EXP (DIV (POW (SUB (x, x_*), 2.0), -1.0)), POW (DIV (POW (2.0, m), FACT (m)), DIV (POW (SUB (x, x_*), LOG (1.0)), 2.0))))
	$f_m(y, x_*)$ = DIV MUL(MUL (POW (DIV (POW (SUB (3.0, 1.0), m), MUL (FACT (m), 1.0)), DIV (1.0, 2.0)), EXP (NEG (POW (SUB (y, x_*), ADD (1.0, 1.0))))), POW (SUB (y, x_*), DIV (m, POW (m, 0.0))))

4.1 ACCURACY OF THE GP SYSTEM

The success rates of the GP system to derive multipole expansions and local expansions for the three different kernels are shown in Figure 9 and Figure 10. The success rate measure is usually defined as the percentage of GP runs terminated with a solution of required quality [16]. But, because in our case the correct solutions for the kernel functions are known (the correct solutions are given in Table 3 and Table 4), here the term "a solution of required quality" refers to the exact solution or the optimal solution. Therefore, the following definition can be used to evaluate the accuracy of the GP system presented in this work:

$$Success\ Rate = \frac{\#of\ runs\ terminated\ with\ the\ \boldsymbol{optimal}\ solution}{total\ number\ of\ runs}$$

For example, a success rate equal to 0.9 implies that GP can find the exact solution for the given kernel in a single run with a probability of 0.9. All values reported in this section are obtained by running GP a number of times (20 runs) and computing the percentage of the total runs that the GP system has found the optimal solution for each given kernel. Figure 9 and Figure 10 show the results of this experiment.

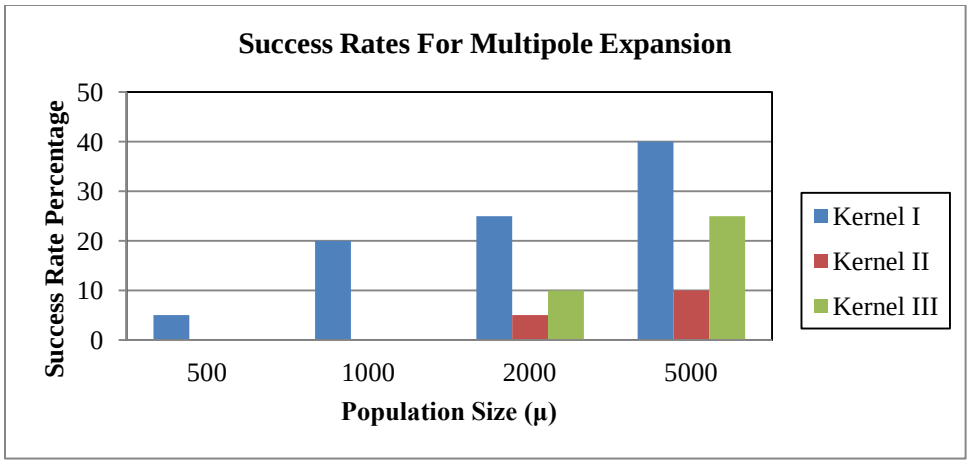

Figure 9 The success rate of GP system in finding multipole expansions for various population sizes (the maximum number of generations is fixed to 50)

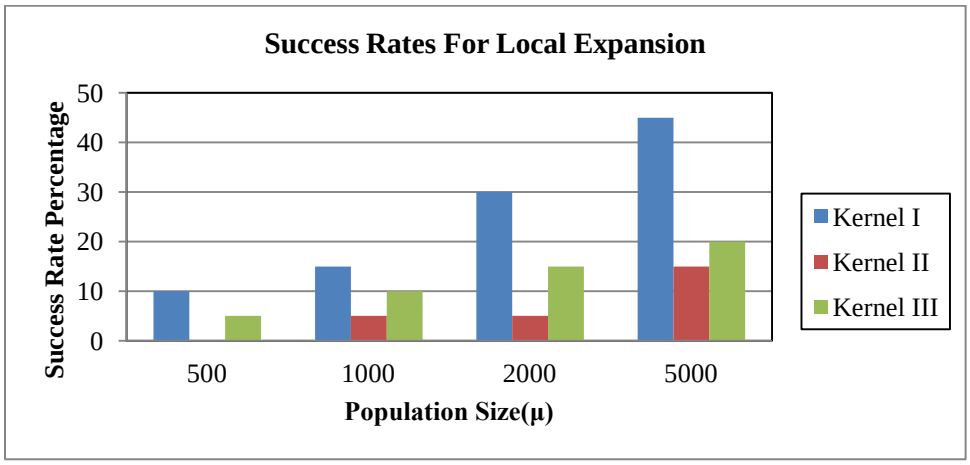

Figure 10 The success rate of GP system in finding local expansions for various population sizes (the maximum number of generations is fixed to 50)

Table 6 Average and standard deviations of the best solutions found by GP in 20 individual runs

	μ = 500		μ = 1000		μ = 2000		μ = 5000	
	MBF	**σ**	**MBF**	**σ**	**MBF**	**σ**	**MBF**	**σ**
Kernel 1	51.43	18.59	73.75	18.19	83.55	14.44	91.65	10.42
Kernel II	17.3	10.68	39.85	19.40	63.78	24.32	72.95	19.57
Kernel III	30.45	21.83	64.83	19.95	73.61	20.93	83.35	14.52

From these figures, it can be seen that finding the analytical expansion for kernel II is much more difficult compared to the other two kernels. This is because the individual corresponding to the solution of this kernel is composed of four tree components as shown in Figure 14, which makes the problem much more complicated in comparison with the other kernels which their corresponding individual is composed of only two tree components. In other words, the search space for kernel II is considerably larger than the search space for the other two kernels. Kernels with four tree components in their individual form the most complicated problems of this kind.

Fortunately, the experiments presented here show that even these types of problems are also solvable using the proposed GP system; however, solving them may require more computational resources. As the problem of factorizing a given kernel, defined in Section 2.1, is a design-type problem (on-off problem) in the sense that it needs to be solved only once and its solution is then used many times in consecutive steps of the simulation by FMM, this amount of computational resources is completely justifiable. Remembering that once the factorization is obtained, FMM reduces the computational complexity of the problem at hand from $O(N^2)$ to $O(N \log N)$, which is a very significant improvement.

Although using success rate to show the accuracy of a GP system can be useful in some cases, but in many other cases using only this measure is not very informative. Therefore, to gain a better understanding, the mean best fitness (MBF) and standard deviation are also reported in Table 6 for each kernel and for various sizes of population (500, 1000, 2000, 5000). Also the corresponding histograms for the three kernel functions are given in Figure 11**,** Figure 12 and Figure 13. In this experiment, we have used the percentage of hits as the fitness measure to be able to compare the results for the three different kernels. The percentage of hits can be defined as the percentage of training data for which the error of the evolved solution is less than a pre-specified amount, say 0.001. For example, it can be seen from Figure 11 that for a population size of 2000, five percent of solutions found by GP system have a hit number between 40 and 60, thirty five percent have a hit number between 60 and 80, and the remaining sixty percent have a hit number greater than 80.

These histograms clearly reveal that beside the complexity of the problem, the low success rates for kernel II are partially related to the definition of success rate which is used here. In practice, as the optimal solution to a given problem is unknown, it is very common to use satisfactory solutions instead of optimal solutions in computing the success rate of a GP system. That way, the success rate of the GP system will be better for all kernels depending on the required quality of the solution. Also, it is also possible to obtain better success rates by letting GP to run for more generations. Again we are faced to a common dilemma which arises in almost any optimization problem: the trade-off between accuracy and time complexity. That is, spending more time will result in solution with better qualities. Here, we have chosen to run GP system for a maximum number of 50 generations because we think that it is enough for the purpose of this paper.

It is worth noting that the GP system presented here can evolve local expansion as well multipole expansion for any given kernel without any change in the system. The only change which is required to switch from multipole expansion to local expansion or vice versa is to replace the fitness cases so that they satisfy the conditions defined in Section 2.1.1 and 2.1.2.

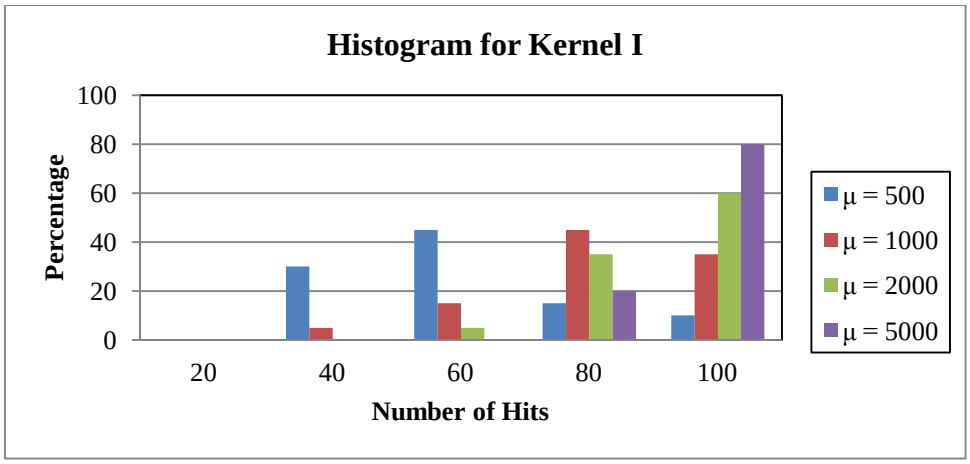

Figure 11 Quality of solutions (in terms of percentage of hits) found by GP system for kernel I.

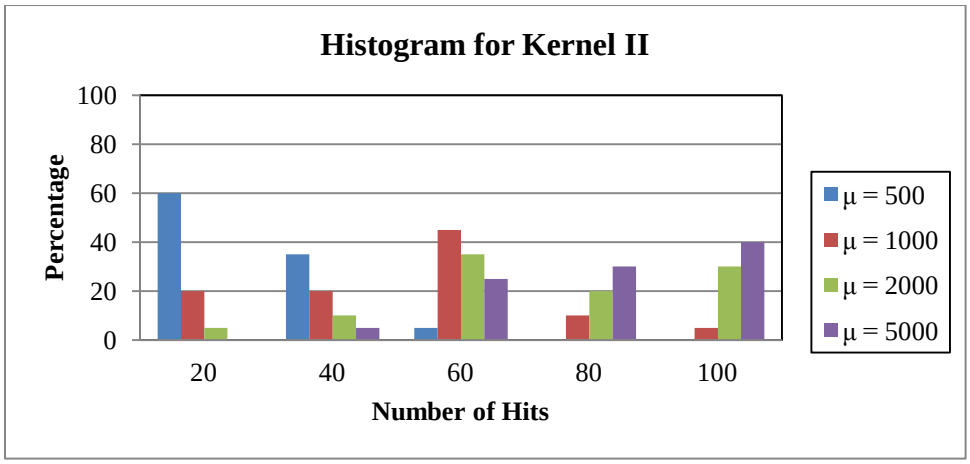

Figure 12 Quality of solutions (in terms of percentage of hits) found by GP system for kernel II.

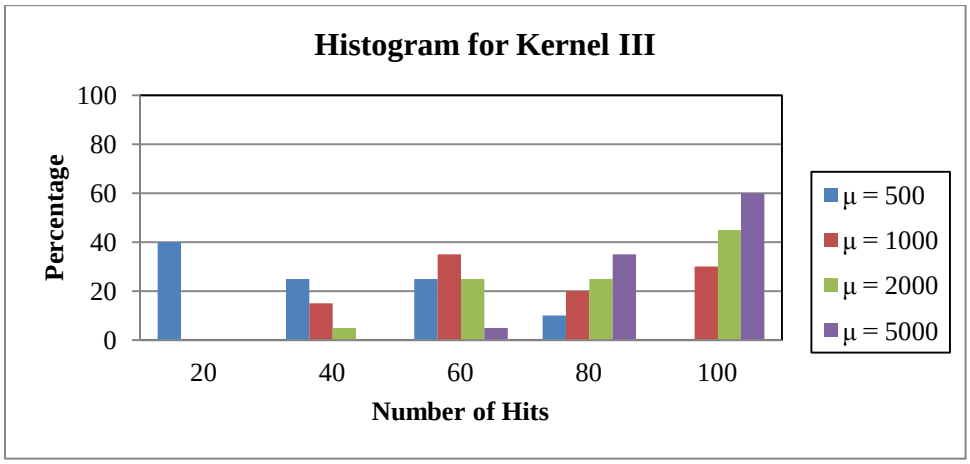

Figure 13 Quality of solutions (in terms of percentage of hits) found by GP system for kernel III.

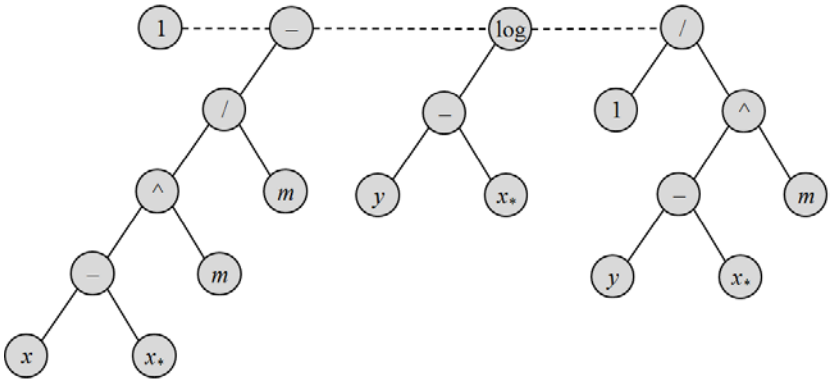

Figure 14 An individual composed of four tree components representing the expansion for $\log(y - x)$

4.2 EFFICIENCY OF THE GP SYSTEM

Here, to have a feeling about the GP system efficiency, we have used the average number of evaluations to a solution (AES) measure. It is computed using the following equation:

$$AES = AGN \times \mu \times n$$

where *AGN* is the average number of generations required to find a solution, μ is the population size and n is the number of fitness cases which are used in the training phase of GP system . The results along the execution times (in terms of second) are presented in Table 7. Again, the results confirm the fact that finding a factorization for kernel II is more complicated compared to the two other kernels. Considering kernel II, it can be seen that the system is about three times faster in finding a solution for kernel I and nearly two times faster for kernel III.

Table 7 Average number of fitness evaluations and average execution times required to find a solution

	AES		**Execution Time (Secs)**	
Kernel	**Multipole expansion**	**Local expansion**	**Multipole expansion**	**Local expansion**
Kernel I	13×10^6	9×10^6	471.23	391.78
Kernel II	41×10^6	37×10^6	1379.11	1197.57
Kernel III	19×10^6	19×10^6	756.93	812.37

Also, it may be helpful to compare these results against the performance of a pure random search. Such a comparison can be helpful in two ways: it may give us a feeling about the difficulty and the complexity of the problem we are trying to solve, and at the same time it can provide us a measure to understand to which extent the proposed system improves the results. To be able to compare the efficiency of random search strategy with GP system, we have given the same amount of time (in terms of number of fitness evaluations required by GP to find an optimal solution) to the random search and then we have recorded the quality of the solutions it has found. The above experiment is repeated for 20 times and the results are reported in Table 8. The values reported in this table represent the number of hits of the best solution found by a Pure Random Search (PRS) in each individual run. Again, a hit occurs whenever the value of the error function defined in (8) is less than 0.001 for a training data. From the table it can be seen that a

pure random search can learn in average 22% of the training data provided in the training set, while with the same amount of time GP can learn all training data correctly. In this experiment, we have used kernel I, which is the simplest one, to show the inefficiency of a pure random search for this problem and also to show the advantage of using GP.

Table 8 of the best solution found by a random search in 20 independent run for kernel I

Run #	**Best solution found by PRS**	**Run #**	**Best solution found by PRS**
1	20	11	16
2	27	12	28
3	14	13	23
4	26	14	9
5	24	15	35
6	6	16	17
7	25	17	16
8	32	18	29
9	31	19	21
10	7	20	28
			Average: 22 %

4.3 CONTROLLING BLOAT

As described in Section 3.6, bloat may have harmful effects on the GP system and thus in any implementation, it is necessary to have some mechanisms to counteract it. These effects are more important when individuals in the GP system can contain several tree structures to represent candidate solutions, like individuals in the proposed GP system in this article. Therefore, it is extremely important to have effective methods to control bloat in this application. To figure out how much the proposed GP system is successful in the fight against bloat, using various anti-bloat techniques discussed in Section 3.6, Figure 15 exhibits the average and maximum size of individuals in a typical run for kernel II. The results in this figure show that the anti-bloat techniques have been successful to control bloat in practice.

5 Summary

In this paper we introduced a genetic programming based learning tool that can be utilized during the design phase of the fast multipole method. The role of the GP system is to derive the multipole and local expansions required in the implementation of the FMM. These analytic expansions vary from kernel to kernel and deriving them manually can be somewhat tedious and a very time-consuming effort, even if such expansions exist. As there is no general technique that can be served to factorize arbitrary kernels, at least at the time of writing this article, hence the GP system proposed here can be regarded as the first automated tool for this purpose. The practical importance of such tool is that it can extend the application domains of the FMM methods to new scientific and engineering domains.

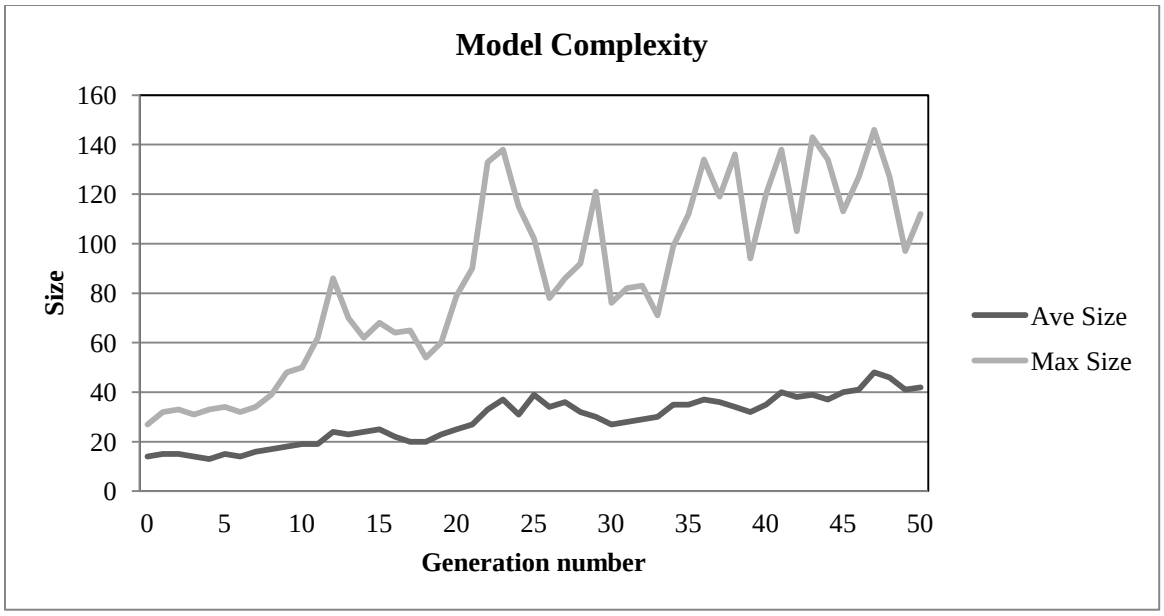

Figure 15 The average and maximum size of individuals in a typical run

One such application would be, for example, the interesting domain of agent based simulations, particularly when there are a very large number of interacting agents to be simulated, with complex interaction rules and patterns. Some examples of such systems can be found in the area of flock simulation, crowd simulation, pedestrian simulation, traffic simulation, and many others. We have implemented a prototype system in Java and tested it on some widely used kernels in the literature. The preliminary results are encouraging and so we hope that the proposed system can be applied successfully to other applications.

References

1. Rokhlin, V., *Rapid solution of integral equations of classical potential theory*. Journal of computational physics, 1983. **60**(2): p. 187-207.
2. Greengard, L. and V. Rokhlin, *A fast algorithm for particle simulations.* Journal of Computational Physics, 1987. **73**(2): p. 325-348.
3. Greengard, L. and V. Rokhlin. *Rapid evaluation of potential fields in three dimensions*. in *Lecture notes in mathematics*. 1988. Berlin: Springer-Verlag.
4. Dongarra, J. and F. Sullivan, *The top ten algorithms of the century*. 2000. **2**(1): p. 22-23.
5. Hanrahan, P., D. Salzman, and L. Aupperle. *A rapid hierarchical radiosity algorithm*. in *SIGGRAPH*. 1991.
6. Singh, J.P., et al., *Load balancing and data locality in hierarchical N-body methods.* Journal of Parallel and Distributed Computing, 1992.
7. Razavi, S.N., et al., *Multi-agent based simulations using fast multipole method: application to large scale simulations of flocking dynamical systems.* Artificial Intelligence Review, 2011. **35**(1): p. 53-72.
8. Razavi, S.N., et al., *Automatic Dynamics Simplification in Fast Multipole Method: Application to Large Flocking Systems*. To be published in Journal of Supercomputing.
9. Ying, L., et al., *A New Parallel Kernel-Independent Fast Multipole Method*, in *The 16th ACM/IEEE Conference on Supercomputing*. 2003, IEEE Computer Society.
10. Ying, L., G. Biros, and D. Zorin, *A Kernel-independent Adaptive Fast Multipole Method in Two and Three Dimensions.* Journal of computational physics, 2004. **196**(2): p. 591-626.
11. Ying, L., *A kernel independent fast multipole algorithm for radial basis functions.* Journal of computational physics, 2006. **213**(2): p. 451-457.
12. Martinsson, P.G. and V. Rokhlin, *An Accelerated Kernel-Independent Fast Multipole Method in One Dimension.* SIAM Journal on Scientific Computing, 2007. **29**(3): p. 1160-1178.
13. Zhang, B., et al., *A Fourier-series-based kernel-independent fast multipole method* Journal of computational physics, 2011. **230**(15): p. 5807-5821.
14. Greengard, L., *The Rapid Evaluation of Potential Fields in Particle Systems*. 1987: ACM press.
15. Koza, J.R., *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. 1992, Cambridge: MIT Press.
16. Eiben, A.E. and J.E. Smith, *Introduction to evolutionary computing*. 1st edition ed. Natural Computing Series. 2003: Springer.
17. Poli, R., W.B. Langdon, and N.F. McPhee, *A Field Guide to Genetic Programming*. 2008.
18. McPhee, N.F. and R. Poli. *Using schema theory to explore interactions of multiple operators*. in *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*. 2002. New York: Morgan Kaufmann Publishers.
19. Langdon, W.B., et al., *The evolution of size and shape*, in *Advances in genetic programming*. 1999, MIT Press: Cambridge. p. 163-190.
20. Soule, T. and J.A. Foster, *Effects of code growth and parsimony pressure on populations in genetic programming.* Evolutionary Computation, 1998. **6**(4): p. 293-309.
21. Crawford-Marks, R. and L. Spector. *Size control via size fair genetic operators in the PushGP genetic programming system*. in *Genetic and Evolutionary Computing Conference, GECCO-2002*. 2002. New York: Morgan Kaufmann Publishers.
22. Langdon, W.B., *Size fair and homologous tree genetic programming crossovers.* Genetic Programming and Evolvable Machines, 2000. **1**(1/2): p. 95-119.
23. Kinnear, K.E. *Fitness landscapes and difficulty in genetic programming*. in *IEEE World Conference on Computational Intelligence*. 1994. Orlando, Florida, USA: IEEE Press.

24. Angeline, P.J. *An investigation into the sensitivity of genetic programming to the frequency of leaf selection during subtree crossover*. in *GECCO '96 Proceedings of the First Annual Conference on Genetic Programming* 1996. Stanford University, CA, USA: MIT Press.
25. Poli, R. *A simple but theoretically-motivated method to control bloat in genetic programming*. in *Proceedings of the 6th European Conference, EuroGP 2003*. 2003. Essex, UK: Springer-Verlag.
26. Zhang, B.T. and H. Mühlenbein, *Evolving optimal neural networks using genetic algorithms with Occam's razor.* Complex Systems, 1993. **7**: p. 199-220.
27. Zhang, B.T. and H. Mühlenbein, *Balancing accuracy and parsimony in genetic programming.* Evolutionary Computation, 1995. **3**(1): p. 17-38.
28. Zhang, B.T., P. Ohm, and H. Mühlenbein, *Evolutionary induction of sparse neural trees.* Evolutionary Computation, 1997. **5**(2): p. 213-236.

Appendix A

Table 9 Numerical Results for experiments in Section 4.1

	Number of Hits											
	μ = 500			**μ = 1000**			**μ = 2000**			**μ = 5000**		
Run	**I**	**II**	**III**	**I**	**II**	**III**	**I**	**II**	**III**	**I**	**II**	**III**
1	39	30	45	64	23	72	61	100	49	79	62	70
2	42	6	23	100	54	87	82	17	68	75	36	100
3	33	10	14	48	22	69	100	87	87	95	98	96
4	46	25	1	82	42	56	88	77	71	67	68	82
5	51	25	50	69	19	60	72	48	64	97	57	84
6	50	15	32	100	69	86	100	42	90	100	71	100
7	48	20	5	78	56	58	86	96	74	98	92	67
8	48	22	14	56	52	55	62	70	53	100	82	78
9	38	8	21	88	15	57	100	67	100	95	57	82
10	67	13	78	68	68	52	87	95	89	100	96	100
11	42	2	15	64	8	62	98	60	96	94	69	95
12	100	14	65	78	42	95	76	34	49	100	47	70
13	31	15	13	83	34	83	100	81	100	89	100	100
14	32	21	16	70	29	69	93	97	89	100	76	65
15	28	5	3	100	42	86	89	68	92	100	100	100
16	73	34	29	48	45	34	100	46	98	77	56	59
17	84	13	51	39	56	28	57	45	54	100	81	98
18	63	25	43	69	48	56	79	47	62	100	97	79
19	56	2	34	71	11	97	68	37	28	81	52	61
20	57	41	57	100	83	34	73	57	59	86	62	81
Mean	**51.40**	**17.3**	**30.45**	**73.75**	**40.9**	**64.8**	**83.55**	**63.55**	**73.6**	**91.65**	**72.95**	**83.35**
S.D.	**18.59**	**10.68**	**21.83**	**18.19**	**20.66**	**19.95**	**14.44**	**24.09**	**20.94**	**10.43**	**19.57**	**14.52**