An Automatic Learning System to Derive Multipole and Local Expansions for the Fast Multipole Method

Seyed Naser Razavi[1], Nicolas Gaud[2], Abderrafiâa Koukam[2], and Naser Mozayani[1]

[1] Iran University of Science and Technology
[2] UTBM
{razavi,mozayani}@iust.ac.ir,
{nicolas.gaud,abder.koukam}@utbm.fr

Abstract. This paper introduces an automatic learning method 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. Later, it was applied successfully in many scientific fields such as simulation of physical systems, Computer Graphics and Molecular dynamics. However, FMM relies on the analytical expansions of the underlying kernel function defining the interactions between particles, which are not always obvious to derive. This is a major factor limiting the application of the FMM to many interesting problems. Thus, the proposed method here can be regarded as a useful tool helping practitioners to apply FMM to their own problems such as agent-based simulation of large complex systems. The preliminary results of the implemented system are very promising, and so we hope that the proposed method can be applied to other problems in different application domains.

Keywords: Agent-Based Simulation, Complex Systems, Fast Multipole Method, Genetic Programming.

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: 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

Y. Tan, Y. Shi, and Z. Ji (Eds.): ICSI 2012, Part II, LNCS 7332, pp. 1–10, 2012.

per iteration. This complexity 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 20th century [4]. FMM can reduce the complexity of evaluating all pairwise interactions in large ensembles of N particles to $O(N\log N)$.

Since its inception, FMM has been applied successfully 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 simulation of large complex systems, when there are millions of interacting agents with complex interaction rules based on physics.

However, the main problem with FMM is that its implementation relies on analytical expansions to approximate 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. 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 [9-11]. These methods are generally known as *kernel-independent* fast multipole methods in the sense that they don't 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, we believe that the most important reason could be related to the fact that these methods are less accurate compared to the FMM, and at the same time they are computationally more expensive. 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. The FMM itself can be used in the simulation of large complex systems consisting of a large number of agents interacting via local rules based on physics such as flocking systems and crowd simulation. Contrary to the kernel independent methods, this approach does not have any negative impact on the efficiency and accuracy of the FMM. 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. 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.

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 $x_i \{1 \le i \le N\}$ in a d-dimensional space (d = 2 or 3). All we need is to compute the potential v_j at M target points $y_j \{1 < j \le M\}$ induced by a kernel K using the following summation:

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

Clearly, a direct implementation of the above summation requires O(MN) operations. For $M = N$, this complexity is quadratic, which is obviously prohibitive for large N. The FMM algorithm can reduce the complexity of the above computations to $O(N log\ N)$, which is a significant reduction, specially for a very large N ($N \ge 10^6$).

FMM achieves its performance by introducing a hierarchical partitioning 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.

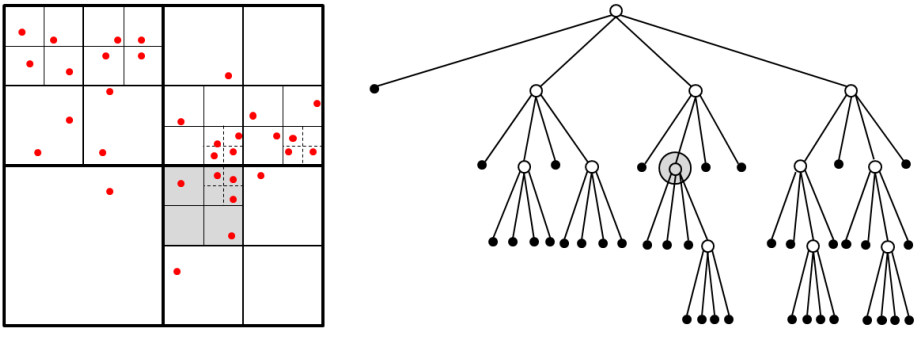

Fig. 1. A 2D particle distribution (left) and its corresponding quadtree (right)

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 above idea in FMM is implemented using expansion operations. In fact, two types of expansions are used in the FMM: the *multipole* 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 of the 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. Now, v_j as defined in (1), can be evaluated in O($pN + pM$) instead of O(MN). For a more detailed description of the FMM algorithm, please refer to [12]. 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 (multipole and local) of any arbitrary kernel, assuming that such expansions exists.

3 Genetic Programming

Genetic Programming (GP), first introduced by Koza [13], uses tree structures 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 is that the results of GP are directly interpretable for humans in contrast to other learning methods such as artificial neural networks.

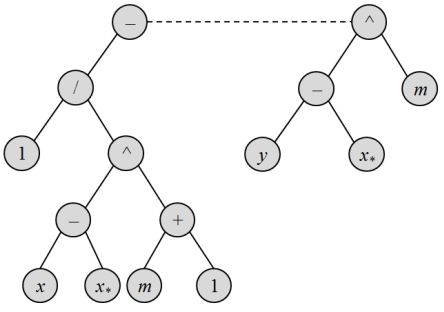

Fig. 2. An individual representing the factorization in (3)

3.1 Model Representation in GP

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 Boolean operators, 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 for $a_m(x, x_*)$ and the other one for $f_m(y, x_*)$. 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. Figure 2 shows an example individual including two trees representing the factorization given in (3).

$$\frac{1}{y-x} = \sum_{m=0}^{\infty} -\frac{1}{(x-x_*)^{m+1}} \cdot (y-x_*)^m \tag{3}$$

3.2 Fitness Function and Selection

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*. In this particular example, each fitness case is an ordered triple in the form of $(x_i, y_i, K(x_i, y_i))$ with x_i and y_i being two randomly selected points in the complex plane which are far enough from each other, and $K(x_i, y_i)$ is the value of the kernel function. 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 ith fitness case is computed as following:

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

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

$$total\ error = \sum_{i=1}^{n} error_i^2 \tag{4}$$

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 one parent from the current population, k (tournament size) individuals are selected at random from the population, and the winner is selected to be a parent.

3.3 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. As individuals in our GP system may consist of several trees, 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, shrink mutation and hoist mutation. These mutations are examples of *fair-size* mutation, as they try to avoid producing very big offsprings during mutation [14]. Based on the suggestions provided in [15], a combination of these mutations is used in our implementation.

Table 1 summarizes the designing parameters of the GP system. 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 almost impossible to make general recommendations for setting optimal values. However, 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.

Table 1. Basic components of the GP system

Objective	Finding the expansions for a given kernel function
Terminal Set	Complex variables x, y, x_*; integer variable m; and random constants chosen from [-5, 5]
Function Set	+, -, ×, /, exp, pow, log, factorial
Fitness	Sum of the squared errors over 100 fitness cases
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
Parameters	See Table 2
Termination	Finding a solution with a total error less than 10^{-5}
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	1000, 5000, 10000
Tournament size	k	50
Crossover rate	p_c	0.90
Mutation rate	p_m	0.01
Generational gap	P_{gap}	0.90
Maximum depth	d_{max}	10
Maximum # of generations	g_{max}	50

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 results reported in this section are obtained by repeating each experiment for 20 times and then averaging the results. 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. Table 3 introduces three frequently used kernels along with

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. 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)) $f_m(y, x_*)$ = DIV (POW (SUB (y, x_*), DIV (m, -1.0)), (SUB (y, x_*))
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))))

their multipole expansions which are used in the experiments. Table 4 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 4 are exactly equal to their corresponding multipole expansion given in Table 3.

4.1 Accuracy of the GP System

The success rate of the GP system to derive multipole expansions for the three different kernels is shown in Figure 3. The success rate measure can be defined as the percentage of GP runs terminated with a solution of required quality [16]. But, because in our case the optimal solutions for the kernel functions are known (see Table 3), here the term "a solution of required quality" refers to the exact solution. Therefore, when it is said that the success rate is equal to 0.9 for kernel I, it means that the GP system has found the optimal solution for this kernel in 90 runs out of 100 runs. These values are obtained by running GP several times and computing the percentage of the total runs that the GP system has found the optimal solution.

Figure 3 shows that finding the analytical expansion for kernel II is much more difficult compared to the other two kernels. This is because the chromosome corresponding to the solution of this kernel is composed of four tree components as shown in Table 3, which makes the problem much more complicated in comparison with the other kernels which their corresponding individual is composed of only two tree components. Obviously, the results could be improved by increasing the population size or the maximum number of generations (or both).

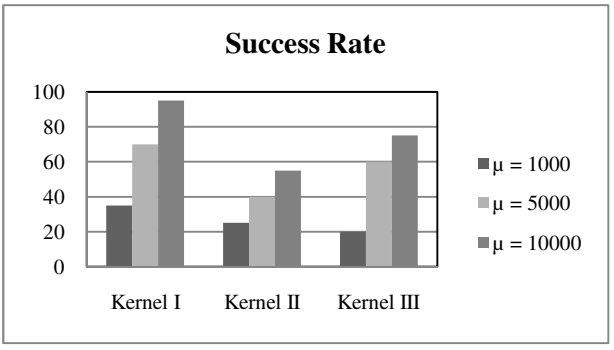

Fig. 3. The success rate of GP system in finding multipole expansions

4.2 Efficiency of the GP System

In this section, we have used the average number of evaluations required to find a solution (AES) to measure GP efficiency. This measure can be computed using the following equation:

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

in which, *AGN* is the average number of generations required to find a solution. The results are presented in Table 5. Again, the results confirm the fact that finding a factorization for kernel II is more complicated compared to the two other kernels.

Table 5. Average number of fitness evaluations required to find a solution

	AES		Execution Time (Secs)	
Kernel	**Multipole**	**Local**	**Multipole**	**Local**
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

5 Summary

This paper introduces a GP-based tool that can be utilized during the design phase of the fast multipole method 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. The practical importance of such tool is that it can extend the application domains of the FMM methods to new scientific and engineering domains, such as agent based simulations (particularly when there are a very large number of interacting agents), flock simulation, crowd simulation, pedestrian simulation, traffic simulation, and many others.

References

1. Rokhlin, V.: Rapid solution of integral equations of classical potential theory. Journal of Computational Physics 60(2), 187–207 (1983)
2. Greengard, L., Rokhlin, V.: A fast algorithm for particle simulations. Journal of Computational Physics 73(2), 325–348 (1987)
3. Greengard, L., Rokhlin, V.: Rapid evaluation of potential fields in three dimensions. Lecture Notes in Mathematics. Springer, Berlin (1988)
4. Dongarra, J., Sullivan, F.: The top ten algorithms of the century 2(1), 22–23 (2000)
5. Hanrahan, P., Salzman, D., Aupperle, L.: 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 35(1), 53–72 (2011)
8. Razavi, S.N., et al.: Automatic Dynamics Simplification in Fast Multipole Method: Application to Large Flocking Systems. To be Published in the Journal of Supercomputing (2012)
9. Ying, L.: A kernel independent fast multipole algorithm for radial basis functions. Journal of Computational Physics 213(2), 451–457 (2006)
10. Martinsson, P.G., Rokhlin, V.: An Accelerated Kernel-Independent Fast Multipole Method in One Dimension. SIAM Journal on Scientific Computing 29(3), 1160–1178 (2007)
11. Zhang, B., et al.: A Fourier-series-based kernel-independent fast multipole method. Journal of Computational Physics 230(15), 5807–5821 (2011)
12. Greengard, L.: The Rapid Evaluation of Potential Fields in Particle Systems. ACM Press (1987)
13. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge (1992)
14. Poli, R., Langdon, W.B., McPhee, N.F.: A Field Guide to Genetic Programming (2008)
15. McPhee, N.F., Poli, R.: Using schema theory to explore interactions of multiple operators. In: GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference. Morgan Kaufmann Publishers, New York (2002)
16. Eiben, A.E., Smith, J.E.: Introduction to evolutionary computing, 1st edn. Natural Computing Series. Springer (2003)