Bridging Scales from Ab Initio Models to Predictive Empirical Models for Complex Materials

W. Michael Brown, Aidan P. Thompson, Jean-Paul Watson, and Peter A. Schultz

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy’s National Nuclear Security Administration under Contract DE-AC04-94AL85000.

Approved for public release; further dissemination unlimited.
Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN  37831

Telephone:   (865)576-8401
Facsimile:   (865)576-5728
E-Mail:      reports@adonis.osti.gov
Online ordering:  http://www.doe.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA  22161

Telephone:   (800)553-6847
Facsimile:   (703)605-6900
E-Mail:      orders@ntis.fedworld.gov
Online order:  http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online
Bridging Scales from Ab Initio Models to Predictive Empirical Models for Complex Materials

W. Michael Brown, Aidan P. Thompson, Jean-Paul Watson, and Peter A. Schultz

Discrete Math and Complex Systems (01412)
Multiscale Dynamic Material Modeling (01435)

Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1316

Abstract

Multiscale materials simulations that capture and quantify the complex dynamical and structural phenomena are crucial to many current and future NNSA and DOE missions. Although simulations are potentially enabled by Tera- and Peta-scale computers and high performance parallel atomistic simulation codes, the lack of adequately predictive atomistic empirical models precludes meaningful simulations for all but a few materials systems. Achieving the goal of using predictive simulations to augment, or even replace, expensive and time-consuming experimental studies requires predictive material-specific simplified empirical models. Herein, we describe the development and validation of the PM-Dreamer software package that is intended to incorporate sophisticated mathematical optimization techniques into the formulation and optimization of robust empirical models. The approach is intended to enable predictive simulations of materials chosen by mission need, rather than dictated by the availability of pre-existing models with sufficient accuracy. We demonstrate the efficacy of the described approach on model systems with atomic configuration energies calculated using known 2- and 3-body interaction potentials.
# CONTENTS

- **Introduction** .................................................................................................................. 7
- **Evolutionary Optimization** .......................................................................................... 8
- **Previous Work** .............................................................................................................. 9
- **Towards the Automated Discovery of Novel Empirical Models** ................................. 10

## Methods

- **Open BEAGLE** ............................................................................................................... 13
- **Distributed Evolutionary Optimization** ......................................................................... 14
- **Hybrid Optimization** .................................................................................................... 16
- **Model Templates** .......................................................................................................... 16
- **Fitness Statistics** ........................................................................................................... 17
- **Vectorization** ................................................................................................................ 18
- **Restarting Optimizations** ............................................................................................. 19
- **Test Cases** .................................................................................................................... 19

## Results

- **Parameter Sets for Optimization** .................................................................................. 21
- **Condensed Phase and Vectorization** ............................................................................ 22
- **Hierarchical Fair Competition** .................................................................................... 23
- **Hybrid Optimization** .................................................................................................... 24
- **Model Templates** .......................................................................................................... 25
- **Correlation and Ordinary Least Squares** ..................................................................... 26
- **The Stillinger-Weber Potential** .................................................................................... 27

## Summary and Future Work

- **References** .................................................................................................................... 31

## Appendix

- **A – Documentation for PM-Dreamer** ............................................................................ 33
- **B – PM-Dreamer Performance for Different Parameter Sets** ....................................... 51

## Distribution

- ......................................................................................................................................... 57
FIGURES

Figure 1. Example of an expression tree used in genetic programming ...................... 9
Figure 2. The concept of Hierarchical Fair Competition ............................................. 15
Figure 3. Correlation vs RMSD for 3 potential solutions .......................................... 18
Figure 4. Optimization efficiency for different system sizes and vectorization .......... 22
Figure 5. Optimization efficiency for random, generational, and HFC strategies .......... 23
Figure 6. Optimization efficiency for different local search probabilities ................... 24
Figure 7. Optimization efficiency using different model templates .......................... 25
Figure 8. Optimization efficiency using the correlation statistic ............................... 26
Figure 9. Summary of performance for the Lennard-Jones test set ............................ 27

TABLES

Table I. Silicon parameters used for Stillinger-Weber potential ............................... 20
Table II. Parameters used for evolutionary optimization ........................................... 21


**Introduction**

Multiscale materials simulations that capture and quantify the complex dynamical and structural phenomena are crucial to many current and future NNSA and DOE missions including:

- radiation damage in electronic devices (QASPR, ELDRS)
- sensitivity of high explosives
- response of materials to high-strain rate dynamical loading
- bubble formation in tritides (neutron generators)
- aging of mixed actinide nuclear fuel elements
- low-melting salts for solar power
- high-activity fuel-cell catalysts
- beyond-end-of-roadmap nanostructured electronic materials

Although simulations are potentially enabled by Tera- and Peta-scale computers and high performance parallel atomistic simulation codes such as LAMMPS, the lack of adequately predictive atomistic empirical models (e.g., interatomic potentials, tight binding models) precludes meaningful simulations for all but a few materials systems. Achieving the goal of using predictive simulations to augment, or even replace, expensive and time-consuming experimental studies requires predictive material-specific simplified empirical models. Such models must incorporate the necessary physics to bridge from more fundamental, but prohibitively expensive, *ab initio* descriptions of materials to more empirical, yet computationally tractable, models.

For many decades, computational efficiency has been achieved in molecular physics with the use of coarse-graining approaches that facilitate a classical description of particle mechanics through dimensionality reduction. While the most common example is the utilization of single particle representations for atoms, investigation into the use of spherical and aspherical particles for representing groups of atoms has recently surged due to the potential for increasing the timescale of nanoparticle, polymer, and protein/DNA simulations [1, 2]. In order to facilitate simulation of a system of particles, it is necessary to obtain equations that calculate the potential energy of a system of particles along with the derivative of the energy with respect to each particle’s position. These derivatives give the particle forces necessary to describe particle motion, perform geometry optimization, etc. Although atomistic and other coarse-grained approaches offer computational efficiency in a straightforward manner, they introduce a fundamental problem: *How do we determine the equations for modeling the instantaneous energy of an arbitrary configuration of particles?*

Traditionally, these equations have been modeled by physicists using a combination of chemical intuition and manual fitting. This approach suffers from two important limitations. First, the models are limited by the observations and time of an individual physicist. Therefore, it is unlikely that these models are optimal in terms of accuracy and transferability to general problems of interest. Second, due to the approximations required for an efficient classical representation of particle mechanics, we cannot expect that a general model exists that is capable of predicting macroscopic properties across the spectrum of particle types and phase space of interest to laboratory
missions. Therefore, in order to achieve a cost-effective capability for particle simulation to support the wide variety of projects important to the success of the laboratory, we require rapid procedures for obtaining models that capture the relevant physics of a given problem.

Herein, we describe research methods to incorporate sophisticated mathematical optimization techniques into the formulation and optimization of robust empirical models. These methods are intended to enable predictive simulations of materials chosen by mission need, rather than dictated by the availability of pre-existing models with sufficient accuracy. We describe the implementation of PM-Dreamer, a software package that offers a unified approach to domain-specific, population-based global/local hybrid optimization algorithms for the identification of models yielding consistently low errors across training data. Finally, we report the validation of PM-Dreamer by obtaining known functional forms for interparticle potentials from particle configuration data.

**Evolutionary Optimization**

In classical particle mechanics, the models typically decompose the system potential energy, $e$, into a set of independent $m$-body interactions that are a function of each particle’s position, $\mathbf{r}$. For a 2-body or pair potential, it is assumed that the energy contributions from each pair of interacting particles are independent of other pairs and therefore,

$$e = \sum_{i=1}^{n} \sum_{j=i+1}^{n} g(\mathbf{r}_i, \mathbf{r}_j).$$

(1)

The force on a given particle is given by the derivative of the energy with respect to that particles position. For a 2-body potential,

$$\mathbf{f}_i = -\sum_{j=1}^{n} \frac{d}{d\mathbf{r}_i} g(\mathbf{r}_i, \mathbf{r}_j).$$

(2)

For a 3-body potential, triplets of atoms are also considered:

$$e = \sum_{i=1}^{n} \sum_{j=i+1}^{n} g(\mathbf{r}_i, \mathbf{r}_j) + \sum_{i=1}^{n} \sum_{j>i} \sum_{k>j} h(\mathbf{r}_i, \mathbf{r}_j, \mathbf{r}_k)$$

(3)

Therefore, for this work, we considered the objective of developing automated methods for obtaining $g$ for 2-body potentials, $g$ and $h$ for 3-body potentials, etc. using training data consisting of particle positions and the corresponding potential energies, $e$. With this approach, we can utilize *ab initio* calculations of the energy of particle configurations to generate the empirical models necessary for efficient calculation of the potential energy surface.

Perhaps the most general approach to symbolic regression of these empirical models is given by genetic programming. In this approach, evolutionary optimization is
utilized to obtain functional forms that fit training data [3]. In genetic programming, a mathematical equation is represented by a tree. For example, the function $xy + |z| + t$ is shown in Figure 1. The tree is evaluated recursively from the root node by applying the operators to each subtree or the constants and variables at the terminal nodes. The search space for the symbolic regression is defined by the possible operators, the variables available for the expression, the set of available constants for the expression, and the maximum depth of the tree. Initially, a random set of individuals (expression trees) is generated to create a population. A fitness metric, typically the root mean square error, is then evaluated using the training data for each expression tree. The evolutionary optimization proceeds by applying operators intended to mimic biological evolution to create a new generation of individuals. In this process, individuals from the current population are selected based on their fitness. For example, a crossover operator can swap subtrees of two individuals selected to breed and a mutation operator can alter a single expression tree by swapping subtrees, deleting subtrees, changing the operator in a node, etc. This process continues until the maximum number of generations has been reached, the maximum amount of time has passed, or an equation has been found that calculates the training data with an error that is below some threshold value. Genetic programming has been utilized for problems ranging from the generation of econometric models to image compression and was chosen for this work due to its potential for finding globally optimal functional forms with little user bias.

**Previous Work**

The most common method employed for obtaining functional forms that describe the potential energy surface in terms of atomic or coarse-grained particles involves the use of analytic fits [4, 5]. In this case, the functional form is “guessed” by physical intuition and the free parameters are optimized to fit available data [6, 7]. Due to the difficulty and time required for obtaining functional forms for analytic fits, interpolation schemes are often employed. In this process, an accurate method such as density-functional theory (DFT) is utilized to provide a sampling of the potential energy surface that facilitates calculation of an approximate continuous potential energy surface. Several methods have been described to accomplish this interpolation including modified Shepard interpolation [8-10], corrugation-reducing interpolation [11-14], and regression with neural networks [6, 15, 16]. These approaches can also be considered as having a fixed functional form defined by the kernel used for interpolation or the neural net. Despite this limitation, they can typically achieve low-error fits to general training data due to the increased parameterization of the model. In trade, the functions generated are typically very large and difficult to interpret. This limitation results in interparticle potentials that are difficult to implement in simulation codes; in addition to energy calculation, it is necessary to use these potentials for calculations of forces, tail-
corrections, etc. Identification and correction of numerical problems in the resulting functions can be difficult. Also, in cases where it is desirable to obtain potentials that are fit in terms of physical constants or other parameters supplied for the different types of particles, it is difficult to interpret how these parameters are utilized in the energy calculation; erroneous terms resulting from overfitting the data are difficult to identify. An additional problem is that the complicated functions add significant computational time to the simulation and are not straightforward to optimize for architecture specific enhancements. In some interpolations approaches, the memory or speed required to compute the energy is dependent on the size of the training data, introducing inefficiency for large training sets and high-dimensional potential energy surfaces. Finally, it is difficult to communicate these models in the scientific literature.

The use of genetic programming (GP) for symbolic regression offers the most general approach, allowing for optimization of the functional form in addition to model parameters. This allows for the generation of compact models without a loss of accuracy. The trade-off is the huge computational effort required for optimization in a search space consisting of possible functions. The use of genetic programming for obtaining models for the potential energy surface was first described by Makarov and Metiu [17]. In their approach, a serial GP implementation was utilized to search for functional forms. In order to obtain low-error results, however, their approach required the use of directed search. In this approach, a significant portion of the functional form is supplied to reduce the size of the search space. Additionally, the authors performed the fitting using the minimum number of particles required to calculate the energy. This allows for a faster optimization and can produce potentials suitable for gas-phase simulations; however, it is unlikely that potentials fit under these conditions would allow for accurate condensed-phase simulations or general purpose applications.

In recent work, we have shown that a parallel GP implementation can successfully obtain the correct functional form for potential energy surfaces using training data sufficient for condensed-phase problems without the use of directed search [18]. For model problems with known solutions, we were able to routinely obtain the exact functional form rather than just low-error fits. In our validation, however, we made use of a relatively simple potential energy function, the search space was limited to integer constants, and the maximum depth for the expression tree was relatively small. For many applications, it will be desirable to make use of more complicated functions in a search space containing more operators and floating-point constants. The addition of new operators and larger expression trees will result in an exponential growth in the size of the search space. This alone is problematic in that the validation runs already required \(~300\) CPU hours each [18].

**Towards the Automated Discovery of Novel Empirical Models**

For many problems of interest, efficient empirical models do not exist or suffer from transferability problems. For example, when attempting to model a semiconductor with a classical approach, simple van der Waals interaction potentials fail are incapable of stabilizing the diamond-like tetrahedral crystal structure of silicon. This can be corrected with the addition of terms to favor the tetrahedral bond angles found in the crystal, as in the Stillinger-Weber potential [19]. When this potential is used, however,
the tetrahedral bias results in inaccurate simulations of amorphous silicon, producing incorrect surface structures and liquid coordination. The more complicated Tersoff potential [20] successfully addressed these issues in silicon. However, when the Tersoff potential was fit to germanium structures, the melting point was severely underestimated [21]. Despite considerable effort, there is no potential capable of accurately modeling melting in germanium.

We have demonstrated a proof-of-principle for the automated generation of compact functional forms necessary for efficient multiscale modeling. In order to find solutions for the types of problems described above, further work is required. To meet this need, we developed a new software package – PM-Dreamer. In this project, we focused on 3 issues important in moving our previous work from proof-of-principal towards an application ready for relevant problems:

1. **Complex Potentials:** The software was designed to allow for the simultaneous optimization of multiple functions involved in 3-body potentials and to calculate the relevant variables from particle configurations with periodic boundary conditions and arbitrary cutoffs.

2. **Efficiency:** A drastic improvement in efficiency is required in order to successfully obtain complicated potentials such as the Stillinger-Weber in a search space with larger expression trees, more operators, and floating-point constants.

3. **Overfitting:** Due to the complex physics and limited training data available for many problems, it is important to mitigate the risk of obtaining erroneous functions that fit only the training points.
Methods

Open BEAGLE

PM-Dreamer was developed in C++ using the Open BEAGLE library for evolutionary computation [22]. This library was developed to meet a need for generic software tools for evolutionary computation that allow for replaceable or modifiable components in an object-oriented framework [23]. The library complies with the C++ ANSI/ISO 3 standard and is licensed under the GNU GPL. The library is designed using an object-oriented architecture with smart pointers for automatic memory allocation management, XML file formats with a built-in parsing facility, parameters and algorithms that are dynamically configurable by files, and a milestone mechanism for evolution recovery and results analysis. The library has several replacement strategies for generation of new populations including generational, steady-state, (mu,lambda), and mu+lambda. The elitism operator is included offering the option to assert that the best individual(s) from a given generation will be present in the next generation. The library supports evolution with multiple populations, multiobjective optimization, population seeding from files, and complete evolution statistics.

For genetic programming, Open BEAGLE uses a standard crossover operator with five mutation operators:

- **Standard** – Standard GP mutation as defined by Koza [3]
- **Swap Node** – Swap nodes in the expression tree
- **Shrink** – Replace a randomly chosen branch with a randomly chosen argument on the branch
- **Swap Subtree** – Swap branches in the expression tree
- **Ephemeral** – Mutate the value of a constant in the expression tree

Three methods can be used for initialization of populations:

- **Grow** – the initial population consists of expression trees with variable depths
- **Full** – the initial population consists of expression trees that all have the maximum depth
- **Half-and-Half** - An equal number of expression trees are generated using a depth parameter that ranges between 2 and the maximum specified depth

Selection of individuals for breeding, etc., can be accomplished using several operators:

- **Random** – Individuals are selected randomly (uniform distribution)
- **Roulette** – Individuals are selected using a proportional roulette selection operator
- **Tournament** – Individuals are selected in a tournament that chooses the best fitness individual from $n$ randomly chosen individuals
- **Parsimony Tournament** – Individuals are selected using a lexicographic parsimony pressure tournament selection based on Luke and Panait.
- **NPGA20p** – Selection for multiobjective optimization
Finally, Open BEAGLE supports automatically defined functions and operators for constrained evolution.

In PM-Dreamer, configuration of the optimization strategy and parameters along with the operators, variables, and constants that compose the search space are specified using an Open BEAGLE supported configuration file that is supplied on the command-line. A full description of the replacement strategies, operators, and parameters for controlling evolution and output of statistics is given in Appendix A.

**Distributed Evolutionary Optimization**

Open BEAGLE has been designed as a serial code with limited support for parallel optimization with multithreaded options only for coevolution. For the domain presented here, this approach is insufficient given the difficulties that arise due to the training data and the complicated search space. Therefore, we implemented support for distributed evolutionary optimization in PM-Dreamer using the Message Passing Interface (MPI) standard [24]. Most existing strategies for parallel optimization with genetic algorithms can be classified into 3 categories [25]. In the first, a master-slave approach is utilized to divide the task of fitness evaluation and/or the application of operators to the individuals in the population. This approach allows for simple load balancing for arbitrary optimization parameters. Of course, there is inefficiency in the method because every individual must be communicated back and forth from the master process at each generation. Additionally, there are limits to the size of a population for performing efficient optimization. In fact, after the population grows past a certain size the optimization efficiency will usually begin to decease. This constraint places severe limitations on the parallel scaling for many problems.

This difficulty can be addressed with the use of multiple populations that evolve independently with some migration of individuals between populations. This approach is generally beneficial for obtaining global minima due to the relatively independent convergence of multiple populations. While this approach can be used to improve the parallel efficiency of master-slave algorithms, it offers an alternative strategy – assigning the multiple populations to different processors. This is the second category and is the most commonly employed parallel approach for genetic algorithms. In this case, the communication is decreased to the migration of a typically small fraction of individuals every specified number of generations. The third category for parallel implementations also uses multiple populations for evolution, but rather than using migration, the approaches utilize a static or dynamic set of overlapping individuals that evolve in more than 1 population.

The probability of obtaining a global minimum can be improved with the use of multiple populations; however, each population will ultimately converge towards some solution resulting in stagnation due to the decrease of variability in a population. Recently, a new evolutionary approach has been proposed to eliminate the premature convergence towards any one solution in evolutionary optimization. The model, called Hierarchical Fair Competition (HFC), is designed to facilitate sustainable evolutionary search by preventing the convergence of a population to the vicinity of any set of optimal
or locally optimal solutions [26]. This is achieved by attempting to ensure a continuous supply of variable genetic material with a hierarchical structure of populations. In this hierarchy, individuals compete with other individuals with similar fitness. Once an individual is obtained with sufficiently high fitness, it moves up the hierarchy to compete with other high fitness individuals. Population sizes are maintained with the introduction of random individuals and/or the decimation of poor fitness individuals. The approach is modeled after advanced social organizations that prevent unfair competition and has been shown to improve the scalability and efficiency of evolutionary algorithms. Although HFC has been evaluated using serial codes, the potential for parallel scalability is obvious.

We implemented two approaches for distributed evolutionary optimization in PM-Dreamer. In the first, multiple populations are used with migration between populations. Each process can have single or multiple populations and the population sizes may vary. The number of individuals and the migration interval can be specified in the configuration file. Here, a set of $n$ populations is divided between the processors (currently, $n$ must be a multiple of the number of processes). In Figure 2, we illustrate the HFC approach using 4 populations. Each population has a rank between 1 and $n$. Rank 1 always has a fitness threshold of 0 and as the rank increases, so should the fitness threshold. Initially, all populations are filled with random individuals. As the evolution proceeds, any individual whose fitness is greater than the threshold for a higher rank population is migrated to that population. Following migration, any population whose size has become greater than the specified fixed population size will decimate the least fit individuals. Any population, whose size is smaller than the fixed size will add randomly generated individuals. Therefore, the rank 1 population has the lowest average fitness. As the evolution process generates individuals with fitness exceeding the lowest threshold,
they migrate to higher rank populations and are replenished by an influx of random individuals. The highest rank population will only accept individuals with high fitness, decimating the least fit individuals whenever new ones arrive.

When one population is allocated to each processor, the arrows in green represent the communication topology for distributed HFC. In order to achieve parallel efficiency, there is a 1-generation lag from the time an individual migrates out of a population on one process to the time it appears in the new population on another. There are two approaches for setting the population thresholds in PM-Dreamer. The first uses an adaptive scheme [27] controlled by a user-specified percentile, $p$. The fitness threshold for a population with percentile $p$ is chosen so that $p$ percent of individuals have equal or lower fitness. In the second, approach, the thresholds are fixed and controlled by specifying the first threshold along with a scaling parameter that is equal to the ratio between fitness thresholds with adjacent ranks. Additionally, the user can specify the interval (in generations) at which HFC migration occurs.

In addition to the distributed optimization strategies, PM-Dreamer extends Open BEAGLE with operators for distributed statistics calculation and support for parallel restarts.

**Hybrid Optimization**

Hybrid optimization, commonly a coupling between global and local search strategies, is not typically employed in genetic programming. The approach has been shown to increase the optimization efficiency in genetic programming for civil engineering problems [28] and is known to be effective when local search strategies are employed in genetic algorithm optimizations. Intuitively, the addition of local search operators would seem to improve the convergence towards correct functional forms with optimization of constants appearing in randomly selected expressions. We therefore implemented a local search operator into PM-Dreamer. The operator utilizes user-specified parameters to control the probability of local search, the maximum iterations of local search, and whether the search optimizes a randomly chosen constant or all constants in the expression tree. Currently, the local search is performed using the derivative-free Nelder-Mead simplex algorithm [29]. In this approach, a starting vector and a step-size are specified to generate the $n+1$ vertices of a simplex for an $n$-dimensional minimization. The simplex moves through the parameter space using a series of geometric transformations including reflection, reflection followed by expansion, contraction and multiple contraction towards the minimum.

**Model Templates**

Function evaluation for particle simulation is complicated by the fact that the instantaneous energy for a configuration is a function of the positions of all of the particles in the system. It would be unacceptably inefficient to attempt to formulate a single function in terms of each individual particle and therefore model templates are provided in PM-Dreamer. Currently templates are available for 2-body and 3-body interparticle potentials. For 2-body potentials, the optimization searches for the function $g$ in equation 1. For 3-body potentials, the functions for $g$ and $h$ in equation 3 are...
simultaneously optimized. This is performed using a single expression tree where the left subtree of the root node represents \( g \) and the right subtree represents \( h \). The training data is supplied as a set of energies, each with a set of particle positions, a cutoff, and an option for periodic boundary conditions. Details on the formats for input files are specified in Appendix A. In future versions, it may be desirable to add additional templates.

Limitations on the amount of training data can create a risk for overfitting in some cases. It has been shown that overfitting can be reduced by fitting not only the functions, but also the derivatives of the functions (i.e. forces) [17]. For many \textit{ab initio} methods, the atomic forces can be easily calculated along with the energy. Therefore, in addition to the templates for energy calculation, PM-Dreamer supports templates that evaluate the particle forces in addition to the system energy, only the particle forces, and only the \( x \)-component of the particle forces. In these cases, forward finite-difference with \( \varepsilon=1\cdot10^{-8} \) is used for force calculation. In cases where both the energy and forces are evaluated, the total fitness is equal to half of the value for the energy fitness statistic plus half of the value for the forces fitness statistic. The fitness statistics are described below.

### Fitness Statistics

It is traditional in genetic programming to use some form of the root mean square error (RMSD) for fitness evaluation in symbolic regression. In PM-Dreamer, this is accomplished using the adaptive RMSD with the fitness given by,

\[
F = \frac{n}{n + \sum_{i=1}^{n} (y_i - \hat{y}_i)^2}, \tag{4}
\]

where \( y \) is the value for the energy or a component of the force in the training set normalized by the number of terms in the summation over \( g \) (see for example Equations 1-3). The normalization is used to prevent any one configuration from being more heavily weighted due to a larger number of computed interactions. \( \hat{y} \) is the value calculated by an individual, also normalized. In this form, the minimum fitness is 0, the maximum fitness is 1, and changes in fitness near 1 are more heavily weighted. The RMSD is a natural choice in that it is an intuitive measure of error. For many regression problems, analytic solutions exist that minimize the RMSD. For genetic programming, however, the RMSD might not be the best choice. Consider a case in which we are optimizing a function \( u(r) \) and the correct solution for the problem is given by \( w(r) \). When we obtain the correct answer, \( u(r) = w(r) \), the data points on a plot of \( u(r) \) vs \( w(r) \) will fall on a straight line with a slope of 1 and an intercept of 0 (blue line in Figure 3). This is the only case in which the RMSD will be 0. Now consider, the case where we have found a very close solution in terms of the search space, \( u(r) = -1\cdot w(r) \). The data points still fall on a line with good correlation (red line in Figure 3), but the RMSD statistic is very poor despite the fact that the expression tree is very similar to the correct answer.
In an attempt to improve optimization efficiency, we implemented an alternative fitness statistic using the Pearson correlation coefficient,

\[
F = \frac{n}{n + 100 - 100 \cdot \text{abs} \left( \frac{1}{n} \sum_{i=1}^{n} \left( y_i - \bar{y} \right) \left( \hat{y}_i - \bar{\hat{y}} \right) \right)},
\]

where \( \sigma \) is used to denote the standard deviation. In this case, the fitness is 1 when there is perfect linear correlation between \( y \) and \( \hat{y} \). In the example in Figure 3, a correct answer would be any function for \( u(r) \) resulting in a straight line on the plot. Any correct function, by this definition, can be altered to result in an RMSD of 0 with rescaling and shifting. This is achieved in PM-Dreamer using ordinary least squares. Use of this statistic is beneficial in that it increases the fraction of the search space corresponding to a correct answer and provides a small decrease in the minimum size of an expression tree required. Presumably, the statistic should also improve efficiency by preventing the early convergence towards functions that do not resemble the correct answer but simply produce results in the vicinity of the training data.

**Vectorization**

Expression evaluation in genetic programming is relatively inefficient when compared to hard-coded expressions because the compiler has no knowledge of the expression that will be evaluated. Each operator in an expression incurs the overhead of a function call because in-lining cannot occur and each expression must be parsed by searching the tree. Additionally, modern processors have the ability to perform simultaneous math operations on a single chip with the use of Single Instruction Multiple Data (SIMD) instructions. These instructions have gone unutilized in current genetic programming codes. This is problematic for our purposes, in that the same expression must be evaluated many times when evaluating the energies and/or forces (see the summations in Equations 1-3).

In order to make energy and force calculation more efficient in PM-Dreamer we added support for vector expressions. When this approach is used, the input for variables in an expression tree is a set of vectors rather than scalar values. Likewise, the answer generated is a vector. This prevents parsing the same tree multiple times because each operator in the expression evaluates the vector in a loop. This also allows for compiler optimizations that utilize SIMD instructions. Vectorization is enabled in PM-Dreamer with a command-line flag that replaces operators and variables with primitives that support vector-vector, vector-scalar, and scalar-scalar operations. It is currently left as an option to aid in compatibility with future Open BEAGLE developments that might not
support vector operations. Additionally, this option can decrease performance when only 1 or 2 terms are evaluated in a given summation.

**Restarting Simulations**

PM-Dreamer offers support for a variety of model templates and fitness statistics. It may be desirable to use different model templates, fitness statistics, and training data during a single run. For example, it might be desirable to start with a small training set utilizing only force calculation in the $x$ dimension for the fitness calculation. This might result in high optimization efficiency. The user might then wish to refine results using a larger dataset with fitnesses that include both the energy and the force. Therefore, we wrote a new restart mechanism that allows the user to restart distributed jobs while changing the optimization parameters, model template, fitness statistic, and/or training data. Upon restart, the fitnesses of all individuals are re-evaluated, statistics are updated, and the hall-of-fame storing the best individuals is recalculated.

**Test Cases**

In order to validate PM-Dreamer and perform efficiency tests, we generated random particle configurations and evaluated the energies and forces using existing interatomic potentials. This differs from the intended application of PM-Dreamer in that we can potentially achieve exact results. In order to evaluate PM-Dreamer performance, training sets were generated for utilization in the optimization along with test sets that were utilized to evaluate performance of a given model on data not used for training. In each case, the performance of each model in calculating the energies and the forces in the test was evaluated.

For 2-body cases, the 12-6 Lennard-Jones potential [30] was utilized for energy and force calculation,

$$
e = \sum_{i=1}^{n} \sum_{j=i+1}^{n} 4\varepsilon \left( \frac{\sigma}{\|r_i - r_j\|} \right)^6 - \left( \frac{\sigma}{\|r_i - r_j\|} \right)^{12} = \sum_{i=1}^{n} \sum_{j=i+1}^{n} 4\varepsilon \left( \frac{\sigma}{r_{ij}} \right)^6 - \left( \frac{\sigma}{r_{ij}} \right)^{12}, \tag{6}$$

where $r_{ij}$ is the distance between each pair of particles. When evaluating the configuration energies, $\sigma$ and $\varepsilon$ were set to 1.0 and a cutoff of 2.5 was used such that no interaction with an interatomic distance greater than 2.5 contributed to the energy. For initial evaluation of the genetic programming parameter space, a set of 10 random configurations including 55-65 pair interactions each were utilized for training. For evaluation of the methods presented in this report, a training set consisting of 5 configurations with 55-65 pair interactions each was utilized. The energies for the 5 configurations were -11.54, -6.16, -9.46, -11.56, and -0.95. For each configuration, the force on a single atomic was evaluated. The forces involved 8-10 interactions. For the test set, 50 random configurations were utilized with 52-104 pair interactions each. The energies in the test set ranged from -21 to 0.52. As with the training set, a single atomic force was calculated per configuration (7-16 interactions per force). The sizes for these datasets were chosen to represent the difficult problem of obtaining models for condensed phases with a small training set.
Using a test set with an order-of-magnitude increase in data points allows us to detect problems with overfitting.

For 3-body cases, the Stillinger-Weber potential [19] was utilized for energy and force calculation (taken from [31]),

\[
E = \sum_i \sum_{j>i} \phi_2(r_{ij}) + \sum_i \sum_{j\neq i} \sum_{k>j} \phi_3(r_{ij}, r_{ik}, \theta_{ijk})
\]

\[
\phi_2(r_{ij}) = A_{ij} \varepsilon_{ij} \left[ B_{ij} \left( \frac{\sigma_{ij}}{r_{ij}} \right)^{12} - \frac{\sigma_{ij}}{r_{ij}} \right] \exp \left( -\frac{\sigma_{ij}}{r_{ij} - a_{ij}\sigma_{ij}} \right)
\]

\[
\phi_3(r_{ij}, r_{ik}, \theta_{ijk}) = \lambda_{ijk} \varepsilon_{ijk} \left[ \cos \theta_{ijk} - \cos \theta_{0ijk} \right]^{2} \exp \left( -\frac{\gamma_{ijk}\sigma_{ij}}{r_{ij} - a_{ij}\sigma_{ij}} \right) \exp \left( -\frac{\gamma_{ik}\sigma_{ik}}{r_{ik} - a_{ik}\sigma_{ik}} \right)
\]

using the parameters for silicon shown in Table I. Energy and force calculation was performed in LAMMPS [32] using a cutoff equal to \(a \sigma\) (Table I). The training set and the test were generated with 10 configurations each with up to 20 pair interactions and 94 3-body interactions. The energies ranged from -15.93 to 6.17. As before, a single atomic force was calculated for each configuration.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\varepsilon)</td>
<td>2.1683</td>
</tr>
<tr>
<td>(\sigma)</td>
<td>2.0951</td>
</tr>
<tr>
<td>(a)</td>
<td>1.80</td>
</tr>
<tr>
<td>(\lambda)</td>
<td>21.0</td>
</tr>
<tr>
<td>(\gamma)</td>
<td>1.20</td>
</tr>
<tr>
<td>(\cos \theta_0)</td>
<td>-1/3</td>
</tr>
<tr>
<td>A</td>
<td>7.04956</td>
</tr>
<tr>
<td>B</td>
<td>0.60222</td>
</tr>
<tr>
<td>p</td>
<td>4.0</td>
</tr>
<tr>
<td>q</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Table I. Silicon parameters used for Stillinger-Weber potential.
Results

Parameter Sets for Optimization

Initially, we performed a rough assessment of efficiency in the genetic programming parameter space given by the replacement strategy and probabilities for crossover and mutation. Based on previous work [3], we assumed that Half-and-Half population initialization with tournament selection would provide the best results. Based on the initial parameters listed in Appendix B, we evaluated the effect of tournament selection size, crossover probability, standard mutation probability, swap mutation probability, shrink mutation probability, subtree-swap mutation probability, and ephemeral (constant) mutation probability. Performance for each parameter was assessed based on the average best fitness using 50 runs each. Each run was performed in parallel on 32 processes and terminated at 2 minutes. The jobs were run on 16 dual 3.4 GHz Intel EM64T processors with an Infiniband interconnect. PM-Dreamer was compiled using the Intel C++ 9.1 compiler with an Open MPI wrapper. The training set used consisted of 10 configurations with 55-65 pair-interactions calculated using the Lennard-Jones potential.

The results from these runs are shown in Appendix B. Based on these results, we selected the initial parameter set shown in Table II. For this work, we pursued evaluation of the generational and HFC replacement strategies. For the runs described in the following sections, the +, -, *, and / operators were included along with pow, log, exp, and abs (absolute value). The variables were interatomic distances for 2-body potentials and interatomic distances and angles for 3-body potentials. Floating point constants were generated between -20 and 20. All primitives had a bias equal to 1.0 such that there was no preference for randomly selecting one primitive over another. The runs in the following sections were all performed on 32 processors for 10 minutes with each data point representing an average over 50 runs.

<table>
<thead>
<tr>
<th></th>
<th>Generational</th>
<th>HFC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Population Size</td>
<td>10000</td>
<td>10000</td>
</tr>
<tr>
<td>Total Populations</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>Tournament Size</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Crossover Prob.</td>
<td>0.9</td>
<td>0.8</td>
</tr>
<tr>
<td>Standard Mutation</td>
<td>0.2</td>
<td>0.15</td>
</tr>
<tr>
<td>Shrink Mutation</td>
<td>0.05</td>
<td>0.05</td>
</tr>
<tr>
<td>Swap Mutation</td>
<td>0.1</td>
<td>0.2</td>
</tr>
<tr>
<td>Constant Mutation</td>
<td>0.1</td>
<td>0.15</td>
</tr>
<tr>
<td>Elitism</td>
<td>1</td>
<td>N/A</td>
</tr>
<tr>
<td>Migration Interval</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Migration Size</td>
<td>500</td>
<td>500</td>
</tr>
<tr>
<td>Initialization Min Depth</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Initialization Max Depth</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Expression Max Depth</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>HFC Interval</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>HFC 1st Threshold</td>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td>HFC Threshold Ratio</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table II. Parameters used for evolutionary optimization
Condensed Phase and Vectorization

In order to assess the impact of system size on optimization efficiency, we performed runs using datasets consisting of 10 configurations, one with 2 pair-interactions per configuration and one with 60. As shown in Figure 4, increasing the system size decreases the optimization efficiency in 2 ways. First, the fitness evaluation is much more expensive; on average, the runs using 2 distances evaluated over 7 times the number of generations than those with 60. This is not the only source of inefficiency, however. We can subtract the inefficiency due to increased computational cost by plotting the average best fitness in terms of the generation instead of time (Figure 4). In this case, the optimization efficiency is still decreased for the larger system size. Presumably, increasing the number of interactions per energy data point increases the optimization difficulty due to error canceling effects. This is relevant because the accurate modeling of condensed phase systems will likely require a large number of interactions per energy calculation. The differences in optimization efficiency might seem mild when viewing the plots in Figure 4; however, it is important to remember that the fraction of the search space corresponding to some range of fitness is expected to decrease sharply as the fitness increases. Therefore, it should often become increasingly difficult to improve individuals with higher fitnesses. It is these improvements that are important in achieving low-error models, however.

![Figure 4](image-url)

**Figure 4.** Optimization efficiency for different system sizes and vectorization. Each data point representations a mean over 50 runs. The mean number of generations evolved in 10 minutes is given on the lower left. RMSD error is calculated for the training set, the test set energies, and the test set forces using the best model obtained at 10 min. The error bar length is equal to the standard deviation.
The impact due to the larger computational expense for fitness evaluation of larger systems can be decreased with the use of vectorization. In this case, vectorization resulted in an approximately 4x speedup when compared to the scalar implementation. Despite this improvement, however, the final models (at the end of 10 minutes) had much larger errors for the \( n=60 \) dataset when evaluated on the test sets (Figure 4). Due to the large speedup achieved, all of the following tests were performed with the use of vectorization.

**Hierarchical Fair Competition**

For the remaining 2-body tests presented, the training and test sets described in the Methods section were used for evaluation. In this case, the training set was decreased to contain only 5 configurations with a much larger test set used to evaluate problems from overfitting. In order to evaluate the efficiency of HFC, we performed optimizations comparing this strategy to a generational strategy with migration and a random strategy. For the random case, all but the best individual are destroyed and replaced with random individuals each generation. As shown in Figure 5, the efficiency of the generational strategy was higher when measured using the training data. Interestingly, HFC ultimately produced better models with much lower errors when evaluated on the test sets. Possibly, this is due to increased survival of general models in intermediate-fitness populations within the hierarchy.

![Graphs showing optimization efficiency for random, generational, and HFC strategies.](image)

*Figure 5. Optimization efficiency for random, generational, and HFC strategies.*
Hybrid Optimization

The efficiency of hybrid optimization was evaluated using generational and HFC strategies with operator probabilities of 0, 0.025, and 0.05. In all cases, the optimization was performed on all constants within an expression with the maximum iterations set to 6. In terms of generational efficiency, hybrid optimization increased the optimization efficiency of both generational and HFC strategies. For generational, an average fitness of 0.92 was obtained in 35 generations with a local search probability of 0.025 as opposed to 100 generations with a local search probability of 0. As shown in Figure 6, this improvement was largely offset by the increased computational time required for local search for the generational strategy. Therefore, more efficient local search methods or derivative-based approaches might be necessary in order to achieve improvements in these cases. For HFC, the improvement in optimization efficiency was significant and nearly half the time was required to obtain a fitness of 0.89 with a local search probability 0.025. Likewise, the final errors on the test sets were approximately half those obtained without local search. Because there was also an improvement in the final errors of the generational strategy for the test sets with local search, the probability of local search was set to 0.025 for the remaining HFC and generational tests.

![Figure 6. Optimization efficiency for different local search probabilities (n).](image-url)
Model Templates

Fitting both the function and the derivative of the function has been shown to reduce overfitting in genetic programming and therefore the approach was implemented into PM-Dreamer. As described in the methods, one atomic force was calculated for each configuration. The force was incorporated into the fitness with a separate evaluation of the fitness statistic for each component of the force. The final fitness was then calculated as one half the fitness for the energy plus one half the fitness for the force. The results when this model template is used are shown in Figure 7 (Energy/Force). For the generational strategy, a drastic improvement is seen with a reduction in the test set errors by almost an order of magnitude. For HFC, the results are also significant with a reduction in error for calculation of the test energies by about 1/3. In addition to reducing overfitting, the approach also resulted in a reduction in training error by preventing convergence towards erroneous functions. Although the convergence is slower, at the end of each run the training error is much lower.

Figure 7. Optimization efficiency using different model templates.

In addition to evaluating the fitness using the energy and force, we also implemented templates to evaluate the fitness using only the force. Our hope was that this would increase optimization efficiency by reducing the number of interactions per data point (see Equation 2 vs Equation 1 and Figure 4). Two approaches were implemented.
One evaluated the error in all 3 components of the force and the other evaluated the error in only the x-component. Because this alters the fitness metric, the first threshold for HFC was adjusted to 0.5 when evaluating only the x-component and 0.3 when evaluating only the force. As shown in Figure 7, this increased the number of generations evaluated in the 10-minute time period, but did not result in a decrease in the test set errors. For the generational strategy, the opposite was true and there was a significant increase in test set errors. Optimization efficiency could potentially be improved by starting an optimization using only forces in the fitness evaluation and then continuing the evolution with the incorporation of energies. This has not yet been tested, however.

Due to the consistently improved results provided by HFC, we continued our efforts using only this strategy with a model template that evaluated the energy and the force as part of the fitness metric. Using these parameters, we tested the impact of the HFC interval on optimization efficiency. The mean RMSD errors for the energies in the test set for a 1, 10, and 20 generation interval were 0.049, 0.028, and 0.029 respectively. We therefore adjusted the HFC interval from 1 to 10.

**Correlation and Ordinary Least Squares**

Finally, we evaluated the use of an alternative fitness statistic for symbolic regression. This was performed using a statistic based on the Pearson correlation coefficient (Equation 5). In this case, it is necessary to rescale the resulting functions obtained by the evolutionary optimization with the use of ordinary least squares. When this statistic is used, the correct functional form is found with certainty within seconds in all 50 runs (Figure 8). The mean RMSD error in calculation of the test set energies is reduced to 5·10⁻⁷.

![Figure 8. Optimization efficiency using the correlation statistic.](image)

We summarize the results from the various methods implemented into PM-Dreamer in Figure 9. By using a generational genetic programming strategy, the mean RMSD error in calculating the test set energies is 1/3 that obtained by a random approach with elitism. By using the HFC strategy, this error is reduced to 1/5 of the previous result. Incorporation of hybrid local search reduces this error by ~1/2. Incorporation of forces into the error evaluation results in ~1/3 the error. Increasing the HFC interval to 10 reduces this error by ~1/2. Finally, use of the Pearson correlation statistic reduces the
error to near zero. With these improvements, a result that required 100 processors for hours can now be obtained on a single-processor desktop in minutes.

![Figure 9. Optimization efficiency using the correlation statistic.](image)

**The Stillinger-Weber Potential**

We applied PM-Dreamer to 3-body problems using energies calculated with the Stillinger-Weber potential to generate the training and test sets described in Methods. These runs were performed on 256 processors for 32 hours. The optimizations were performed using HFC configured with the parameters in Table II with the exception that the maximum expression tree depth was set to 10. Hybrid local search was performed with a 0.025 probability and no more than 6 iterations. The 3-body energy/force model template was used with the correlation fitness statistic. The variables were the interatomic distances and angles formed by all triplets of atoms. In preliminary runs, we were able to achieve low-error fits to the Stillinger-Weber equation, but not exact matches. For the test set energies (normalized), we obtained an RMSD of 0.010. However, the resulting expressions were very large (>150). By implementing a fitness penalty of 0.05 for each node over a maximum size of 50, we were able to reduce the expression size (to 50). In this case, the mean RMSD in calculation of the normalized energies (over 4 runs) was 0.0182. This corresponds to a correlation coefficient of 0.9995 between the calculated energies and the training values with a <1% error in calculation of configuration energies.
Summary and Future Work

We developed PM-Dreamer for the automated discovery of novel empirical models for particle mechanics. In this project, we focused our research efforts on modifications to facilitate a drastic improvement in optimization efficiency when compared to a canonical genetic programming framework. We described approaches for distributed evolution, hybrid optimization, vectorization, and improved fitness metrics. We obtained an improvement in efficiency of several orders of magnitude with the use of a Hierarchical Fair Competition algorithm, hybrid optimization, vectorization, and a correlation-based fitness statistic. The use of the Pearson correlation coefficient with ordinary least squares rescaling resulted in a drastic improvement and it is our position that this or a similar approach should be the standard for genetic programming symbolic regression problems. We have shown that the utilization of particle forces along with the system energy can reduce overfitting and ultimately produce more accurate models by preventing convergence towards erroneous functions. Interestingly, our results also suggest that Hierarchical Fair Competition can reduce problems from overfitting when there is limited training data. With these advancements, we are now able to obtain low-error fits for complicated models such as the Stillinger-Weber potential.

After further validation and optimization for 3-body potentials, we will have made sufficient advances in this project to allow for the application of PM-Dreamer to real problems. As a next step, we will apply PM-Dreamer to the problem of modeling germanium. This work will consist of DFT calculations to obtain training and test data, automated model development, and implementation into simulation codes. Further model validation can then be performed with the accurate calculation of observable macroscopic properties. Although it will not be part of our immediate efforts, PM-Dreamer should also be beneficial for problems in coarse-graining. For example, it can be shown that the Lennard-Jones potential does not provide a correct scaling of energy with distance when using coarse-graining to model large particles [2]. A potential correcting for this problem has been derived using a continuum approximation that integrates the Lennard-Jones interaction over both particles. However, when modeling a complex mesogen with a single particle, it seems likely that using a different functional form altogether may produce more accurate results.

In addition to increased efficiency, PM-Dreamer has been developed to facilitate the automated discovery of complex potentials. In this work, we implemented a capability for simultaneous optimization of the multiple functions in 3-body potentials, calculation of particle forces, and the calculation of descriptors such as interparticle distances and angles. For future work, it will likely be desirable to generalize this further with the addition of alternative model templates and descriptors such as bond order, number of neighbors, etc. This will potentially allow for improved accuracy by increasing the number of variables available for the optimization (and decreasing user-bias). An additional difficulty in genetic programming that is not suitably accounted for in PM-Dreamer is the generation of “dead” subtrees that have no impact on the energy calculation. For example, the branch “*[(1-1+1)^1+x^2*0]” has no impact on the function calculation. These “dead” subtrees can take on complicated forms that are difficult to interpret or identify. Their occurrence in genetic programming has been likened to the fact that only a small percentage of DNA actually encodes proteins [3]. PM-Dreamer
offers some mechanism to counter this with the use of selection strategies that favor parsimony and fitness operators that add penalties for large trees. A more elegant strategy can be implemented with the use of a simplification operator that replaces a subtree with a constant if there is little to no variation across the training data. Application of this operator at some generation interval might also improve efficiency by shrinking the average expression size.

Increasing optimization efficiency will always be of interest in PM-Dreamer. While increasing the amount of training data leads to a constant increase in computational cost, increasing the dimensionality of the search space can have a devastating impact. The use of forward finite difference for force calculation and derivative-free local search is one target for efficiency improvement in PM-Dreamer. Automatic differentiation codes such as Sacado allow for the calculation of accurate forces during energy calculation and facilitate the use of derivative-based optimization approaches. The use of automatic differentiation in PM-Dreamer can therefore potentially decrease the time required for derivative calculation and improve the efficiency of hybrid local search. When performing simultaneous optimization on multiple functions, PM-Dreamer uses a single expression tree allowing crossover and mutation to occur across functions. Further improvements in optimization efficiency might be obtained by asserting independent evolution of each function in the expression trees in terms of crossover and mutation.
References


Appendix A - Documentation for GP-Force

NAME

pm_dreamer - Optimization of functional forms for particle mechanics.

VERSION

Version 0.1

SYNOPSIS

pm_dreamer input_data beagle_config_file [-c] [-f file_header] [-g start_size end_size] [-h] [-n notice_level] [-r rseed] [-s stat_type] [-t energy_type] [-z]

DESCRIPTION

PM-Dreamer is software for generation of empirical models for particle mechanics. The software takes as input data from a series of particle configurations and the corresponding energies and/or particle forces associated with those configurations. The output from PM-Dreamer is a set of functions that can potentially be used to calculate configuration energies for particles giving the force-field necessary for particle simulations.

PM-Dreamer obtains the equations for energy calculation using a combination of genetic programming and local search in order to minimize the root-mean square error in the calculation of energy and/or particle force. The genetic programming is based on the Open-BEAGLE library for evolutionary computation. This library has been extended in PM-Dreamer to allow for massively parallel optimization, hybrid local search, vectorized expression evaluation, template-based evaluation of fitnesses using particle configurations with periodic boundary conditions for 2 and 3-body particle interactions, and parallel restarts with the capability to switch datasets and/or function templates.

The input for PM-Dreamer consists of the input_data file that contains particle configuration data and the beagle_config_file that facilitates parameterization of the optimization. The formats available for the input_data are described in the Fitness Evaluation section and can be specified with the -t flag. The format for the beagle_config_file is taken from Open Beagle with the extensions described throughout this documentation. Examples for both should have been included with the software package.

There are three types of output for PM-Dreamer. Console output describes the progress and statistics of the run and is controlled with the -n flag. Log file output also describes statistics in XML as specified in the beagle_config_file. The default filename for the log file for serial runs is gp_force.log. For parallel runs a separate log file is written for each process with the default name gp_force_RR.log where RR is the process rank. The final output format consists of milestone files. These files contain an XML description of all of the expressions at a given point in the optimization and are also used to restart runs. The default name for the milestone files is gp_force.obm for serial or gp_force_RR.obm for parallel runs. Utilities for generating plots from the log files and graphic representations of expressions should have been included with this software package.

The following definitions are used throughout the documentation:
Individual A single mathematical expression for calculation of the energies of particle configurations.

Primitive A primitive is a node in the expression tree. Examples of primitives include unary and binary mathematical expressions, variables used to describe the particle configurations, and constants in the expression.

Terminal/Constant/Ephemeral A terminal is a primitive that takes no arguments. A constant/ephemeral is a terminal that is not used as a variable in the expression. Constants are typically randomly generated and can change during the optimization by mutation to generate a new random number or by local search executed to optimize the constants in an expression. Constants that should not be modified (such as pi) can also be specified.

Fitness A metric describing the error in an Individual’s calculation of the energy using the training data.

Population/Deme A population or deme is a group of individuals that evolve together. Crossover occurs using multiple individuals from a population and selection occurs based on the individuals in a single population. Multiple populations can be used in a run. The populations evolve separately, but can interact through migration of individuals between the populations.

Island Here, an island is used in parallel runs to describe the population or set of populations undergoing evolution in a single MPI process.

Vivarium All of the populations involved in a run.

Hall of Fame The Hall of Fame contains the n individuals with the best fitness(es) found during a run.

Milestone/Restart File These files contain the output of all of the individuals at some point in the optimization and have the extension .obm

Hybrid Optimization/Local Search Hybrid optimization occurs separately from the evolutionary optimization. With a specified probability, local search is performed on an individual to optimize 1 or multiple constants.

PARAMETERS

-c Restart from existing milestone files. When restarting, the functional form (-t), the beagle_config_file file, and the input_data can be different from those used in the original run. This allows the user to change parameters and/or add data to refine runs. If the -f flag was used to specify a non-default file header for the restart files. The -f flag should also be specified again with the same name when using -r. When running in parallel, the same number of processes should be used for the restart. If a smaller number is used, the extra individuals will be ignored. If a larger number is used, an error is generated. When restarting an optimization, the generation number starts at the last generation in the milestone file. Therefore, the maxgens termination criterion may need to be increased. The restart files are read by the ReadRestartOp in the beagle_config_file

-f file_header
Specify the header for the .log output file and the .obm milestone files. The default is gp_force.

-g start_size end_size

Scale the fitness by the number of nodes in the tree. This can be used to reduce the average size of individuals. A tree with start_size or smaller nodes has a maximum fitness of 1.0. A tree with end_size or greater nodes has a maximum fitness of 0.0.

-h Print out the man page for help

-n notice_level

Set the degree of program output. Use:

- n 0 No output
- n 10 Normal program output
- n 20 Parameters useful for reproducing the results
- n 30 All output. The degree of Open Beagle Output changes at 10,20, and 30.

-r rseed

Specify the random seed (unsigned long). Default is 1.

-s stat_type

Choose the fitness statistic used. Options are RMSD for adaptive RMSD, CORR for the Pearson correlation coefficient, and OLS for ordinary least squares fitting. See the Fitness section for details on each method.

-t energy_type

Specify the functional form of the energy function

-z Use vectorized tree evaluation. This will typically be at least 4x faster for the optimized Beagle library and greater than 15x faster for the debug library.

BEAGLE CONFIGURATION FILE

The Beagle configuration file is used to control the optimization including the functions, terminals, operators, and replacement strategies that are used. Details on each section follow. A template for a configuration files is:

<?xml version="1.0" encoding="ISO-8859-1"?>
<Beagle>
  <Evolver>
    <BootstrapSet>
      ... Population Initialization ...
    </ BootstrapSet>
  </Evolver>
</Beagle>
... Fitness Evaluation ...
... Mutation ...
... Selection ...
... Selection ...
... Migration ...
... Statistics ...
... Termination Criteria ...
... Restart File Output ...

</MainLoopSet>
</Evolver>
<System>
<PrimitiveSuperSet>
<PrimitiveSet>
... Functions ...
... Terminals ...
</PrimitiveSet>
</PrimitiveSuperSet>
<Register>
... Register values ...
</Register>
</System>
</Beagle>

**POPULATION INITIALIZATION**

The initialization is accomplished using the following operators:

**GP-InitHalfOp**

Koza’s ramped half-and-half generative method. An equal number of expression trees are generated using a depth parameter that ranges between 2 and the maximum specified depth.

**GP-InitFullOp**

The initial population will consist of expression trees that all have a depth equal to the maximum depth.

**GP-InitGrowOp**

The initial population consists of expression trees of variable depths.

**RestartReadOp**

Read in population from a restart (milestone) file. This operator replaces the MilestoneReadOp operator in Open Beagle to allow the parameters in the beagle_config_file to override those in the milestone file. The example below checks to see if the register, ms.restart.file, is set. If it is, a restart file is read in. Otherwise, a population is generated using half-and-half followed by fitness evaluation and statistics output:
<BootStrapSet>
  <IfThenElseOp parameter="ms.restart.file" value=""/>
  <PositiveOpSet>
    <GP-InitHalfOp/>
    <EnergyOp/>
    <GP-StatsCalcFitnessSimpleOp/>
  </PositiveOpSet>
  <NegativeOpSet>
    <RestartReadOp/>
  </NegativeOpSet>
</IfThenElseOp>
</BootStrapSet>

**REPLACEMENT STRATEGY AND MIGRATION**

The replacement strategy is specified using the follow operators:

**DecimateOp**

Shrink the population size by keeping the n best individuals

**GenerationalOp**

Breeding tree following a generation by generation replacement strategy

**HierarchicalFairCompetitionOp**

HCF operator inspired by the work of Hu and Goodman

**MigrationRandomRingOp**

Migrate randomly chosen individuals between populations using a ring topology

**MuCommaLambdaOp**

A (Mu,Lambda) operator generates Lambda children individuals from a population of Mu parents (where Lambda > Mu). From these Lambda individual, it keeps the Mu best to constitute the new generation.

**MuPlusLambdaOp**

A (Mu+Lambda) operator generates Lambda children individuals from a population of Mu parents (usually where Lambda > Mu). From the Mu parents and the Lambda individual, it keeps the Mu best individuals to constitute the new generation.

**NSGA2Op**

The NSGA2 replacement strategy implement the elitist multiobjective evolutionary algorithm NSGA2 (Non-dominating Sorting Genetic Algorithm)

**OversizeOp**

An oversize operator generates (ratio * population size) children individuals from a population of Mu parents.

**SteadyStateOp**

Steady state replacement strategy operator
**CROSSOVER AND MUTATION**

Crossover and mutation are specified using the following operators:

**GP-CrossoverOp**

- Crossover of two individuals to produce a new individual

**GP-MutationEphemeralDoubleOp**

- Mutate the value of a randomly chosen double precision constant in the tree

**GP-MutationShrinkOp**

- Replace a randomly chosen branch with a randomly chosen argument on the branch

**GP-MutationStandardOp**

- Canonical GP Mutation

**GP-MutationSwapOp**

- Swap nodes in the expression tree

**GP-MutationSwapSubtreeOp**

- Swap branches in the expression tree

**SELECTION**

Selection is specified using the following operators:

**NPGA20p**

- Multiobjective evolutionary algorithm NPGA 2 (Niched Pareto Genetic Algorithm)

**SelectParsimonyTournOp**


**SelectRandomOp**

- Select an individual in a population randomly operator class (uniform distribution).

**SelectRouletteOp**

- Proportional roulette selection operator class.

**SelectTournamentOp**

- Tournament selection operator class.

**TERMINATION**

Optimization is terminated using the following operators:
TermMaxGenOp

Maximum generation termination criterion operator.

TermMaxFitnessOp

Maximum fitness value termination criterion operator class.

TermMaxHitsOp

Number of hits required in an individual in order for the evolution process to terminate.

TermMaxEvalsOp

Maximum number of fitness evaluations termination criterion operator.

RESTART FILES AND POPULATION OUTPUT

Files output containing populations that can also be used for continuing a simulation are generated with the following operators. (See also POPULATION INITIALIZATION.)

MilestoneWriteOp

Write out a milestone file

ParetoFrontCalculateOp

Evaluate Pareto front from demes and vivarium and put it in place of the actual hall-of-fame. The Pareto front is evaluated just before milestones are written. If previous hall-of-fame are presents in the demes/vivarium, they are erased. This operator must be in the evolver’s operator sets between the termination criterion check operators and the MilestoneWriteOp operator.

STATISTICS

Statistics on fitness, function and terminal usage, and expression tree size are generated using the follow operators:


ADF and Constrained Operators

Automatically Defined Functions (ADF) and constrained operators are also available:


The additional primitives for the ADF operators include:

ADF (Automatically Defined Function) and ARG (Generic Argument for ADF)
FUNCTIONS

The following functions can be utilized as primitives in the expression

Abs, Add, Cos, Divide, Exp, Log, Multiply, Sin, Subtract

Additional functions added by PM-Dreamer are described below. Functions are added by listing the function name and bias in the primitive set. For example:

<Primitive name="ADD" bias="1"/>

TERMINALS

The terminals are primitives in the expression tree that do not take arguments (e.g. constants in the expression or variables of the expression). Some that can be included are a double precision number [-1, 1] (E), Pi (Pi), and/or a variable, (X), for the potential:

<Primitive name="E" bias="1"/>
<Primitive name="Pi" bias="1"/>
<Primitive name="X" bias="1"/>

ADDITIONAL PRIMITIVES

The additional function and terminal primitives have been added:

E_i  
Double precision integer [-20, 20]. Generation or mutation of E_i results in an integer, however, hybrid optimization can produce non-integer numbers.

E_d  
Double precision number [-20, 20].

Pow
Exponentiation.

REGISTERS

The registers allow for parameterization of the operators and optimization (e.g. mutation frequency, number of generations, etc.). The registers can be set by specifying the register and the value in the configuration file:

<Entry key="ec.pop.size">500/500/500/500</Entry>
<Entry key="ec.term.maxgen">100</Entry>

A list of registers and short descriptions is given below. If the value type of a register begins with U, the type is unsigned. If the value type is an array, individual elements are delimited using a /.

ec.conf.dump <String> (def: "")
Filename used to dump the configuration. A configuration dump means that a configuration file is written with the evolver (including the composing operators) and the register (including the registered parameters and their default values). No evolution is conducted on a configuration dump. An empty string means no dump.
ec.elite.keepsize <UInt> (def: 1)

Number of individuals keep as is with strong n-elitism.

ec.hof.demesize <UInt> (def: 0)

Number of individuals kept in each deme’s hall-of-fame (best individuals so far). Note that a hall-of-fame contains only copies of the best individuals so far and is not used by the evolution process.

ec.hof.vivasize <UInt> (def: 1)

Number of individuals kept in vivarium’s hall-of-fame (best individuals so far). Note that a hall-of-fame contains only copies of the best individuals so far and is not used by the evolution process.

ec.init.seedsfile <String> (def: "")

Name of file to use for seeding the evolution with crafted individual. An empty string means no seeding.

ec.mig.interval <UInt> (def: 1)

Interval between each migration, in number of generations. An interval of 0 disables migration.

ec.mig.size <UInt> (def: 5)

Number of individuals migrating between each deme, at each migration.

ec.pop.size <UIntArray> (def: 100)

Number of demes and size of each deme of the population. The format of an UIntArray is S1,S2,...,Sn, where Si is the ith value. The size of the UIntArray is the number of demes present in the vivarium, while each value of the vector is the size of the corresponding deme.

ec.repro.prob <Float> (def: 0.1)

Probability that an individual is reproduced as is, without modification. This parameter is useful only in selection and initialization operators that are composing a breeder tree.

ec.sel.tournsize <UInt> (def: 2)

Number of participants for tournament selection.

ec.term.maxfitness <Float> (def: 1)

Fitness value to reach before stopping evolution.

ec.term.maxgen <UInt> (def: 50)

Maximum number of generations for the evolution.

gp.cx.distrpb <Float> (def: 0.9)

Probability that a crossover point is a branch (node with sub-trees). Value of 1.0 means that all crossover points are branches, and value of 0.0 means that all crossover points are leaves.
gp.cx.indpb  <Float> (def: 0.9)

Individual crossover probability at each generation.

gp.init.maxargs  <UIntArray> (def: 0/2)

Maximum number of arguments in GP tree. Tree arguments are is usually useful with ADFs (and similar stuff).

gp.init.maxdepth  <UInt> (def: 5)

Maximum depth for newly initialized trees.

gp.init.maxtree  <UInt> (def: 1)

Maximum number of GP tree in newly initialized individuals. More than one tree is usually useful with ADFs (and other ADx).

gp.init.minargs  <UIntArray> (def: 0/2)

Minimum number of arguments in GP tree. Tree arguments are is usually useful with ADFs (and similar stuff).

gp.init.mindepth  <UInt> (def: 2)

Minimum depth for newly initialized trees.

gp.init.mintree  <UInt> (def: 1)

Minimum number of GP tree in newly initialized individuals. More than one tree is usually useful with ADFs (and other ADx).

gp.mutephdbl.indpb  <Float> (def: 0.05)


gp.mutephdbl.primit  <String> (def: E)


gp.mutshrink.indpb  <Float> (def: 0.05)

Shrink mutation probability for an individual. Shrink mutation consists in replacing a branch (a node with one or more arguments) with one of his child node. This erases the chosen node and the other child nodes.

gp.mutstd.indpb  <Float> (def: 0.05)

Standard mutation probability for an individual. A standard mutation replaces a sub-tree with a randomly generated one.

gp.mutstd.maxdepth  <UInt> (def: 5)

Maximum depth for standard mutation. A standard mutation replaces a sub-tree with a randomly generated one.

gp.mutswap.distrpb  <Float> (def: 0.5)

Probability that a swap mutation point is a branch (node with sub-trees). Value of 1.0 means that all swap mutation points are branches, and value of 0.0 means that all swap mutation points are leaves. Swap mutation consists in exchanging the primitive associated to a node by one having the same number of arguments.
gp.mutswap.indpb  <Float> (def: 0.05)

Swap mutation probability for an individual. Swap mutation consists in exchanging the primitive associated to a node by one having the same number of arguments.

gp.tree.maxdepth  <UInt> (def: 17)

Maximum allowed depth for the trees.

gp.try  <UInt> (def: 2)

Maximum number of attempts to modify a GP tree in a genetic operation. As there is topological constraints on GP trees (i.e. tree depth limit), it is often necessary to try a genetic operation several times.

lg.file.level  <UInt> (def: 3)

Log level used for file output generation. Log levels available are: (0) no log, (1) basic logs, (2) stats, (3) general informations, (4) details on operations, (5) trace of the algorithms, (6) verbose, (7) debug (enabled only in full debug mode).

lg.show.class  <Bool> (def: 0)

Flag whether class name is outputed in the logs.

lg.show.level  <Bool> (def: 0)

Flag whether logging level in outputed in the logs.

lg.show.type  <Bool> (def: 0)

Flag whether message type is outputed in the logs.

ms.restart.file  <String> (def: "")

Name of the milestone file from which the evolution should be restarted. An empty string means no restart.

ms.write.compress  <Bool> (def: 1)

If true, this flag indicates that milestones will be compressed with gzip. Otherwise, each milestone are kept plain text.

ms.write.interval  <UInt> (def: 0)

Milestone saving interval (in number of generations). When zero, only the last generation milestone is saved.

ms.write.over  <Bool> (def: 1)

If true, this flag indicates that old milestones should be over-written. Otherwise, each milestone has a different suffix.

ms.write.perdeme  <Bool> (def: 0)

If true, this flag indicates that separate milestones should be written after each demes processing. Otherwise milestones are written after the processing of a complete populations.
**FITNESS EVALUATION**

The fitness evaluation in PM-Dreamer can be calculated using several different fitness statistics specified with the \(-s\) flag. The fitness in each case is given by \(F\):

\[
F = \frac{1}{1 + c \times s(e,p)}
\]

where \(e\) represents the energies and/or forces from the training set normalized by the number of distances used in the calculation of each energy/force and \(p\) represents those normalized values as calculated by a candidate individual. For the adaptive RMSD, \(c=1\) and \(s(e,p)\) is the root mean squared error between \(e\) and \(p\). For the Pearson correlation coefficient, \(c=100\) and \(s\) is given by the absolute value of the correlation coefficient between \(e\) and \(p\). For OLS, ordinary least squares is performed to give the linear rescaling of \(p\) that results in the lowest RMSD with \(e\). In this case, \(s\) is this RMSD and \(c\) is 1. The calculation of \(p\) according to the candidate expression is performed using one of several templates specified with the \(-t\) option. For all, the fitness calculation in the `beagle_config_file` file is specified using `EnergyOp`.

**PAIR POTENTIALS** \((-t\ pair)\)

The pair potential, `pair`, is the default functional form used for fitness calculation. It is calculated as:

\[
p = \frac{1}{\eta} \sum_{i} g(X_i)
\]

where \(X_i\) is a single variable describing the particle pair (e.g. the inter-particle distance) and \(g\) is the function optimized using genetic programming. The fitness of the function is evaluated using a set of sample configurations for which the energies have been calculated. The format for the input file is:

```plaintext
# Comments for the input file
e X1 X2 ...
e X1 X2 ...
...
```

Each line begins with an energy \(e\) and is followed by a variable number of data points for each pair in the configuration. Empty lines and lines beginning with \# are ignored. In order to use this template, the \(X\) variable should be added to the primitive set:

```xml
<Primitive name="X" bias="1"/>
```

**PAIR POTENTIALS WITH FORCE** \((-t\ paird)\)

The pair potential with force, `paird`, is similar to `pair` with the exception that a particle force is supplied for a particle in each configuration allowing the potential function to be fit to both the energy and the force. When this style is used the fitness is one half the fitness statistic calculated for the energies plus one half the fitness statistic calculated for the forces.

Here, the potential is calculated as described for the `pair` style, and the force is calculated as the negative gradient of the energy for a particle using forward finite-difference. The format for the input data
file is:

# Comments for the input file
cutoff $C$
e $f_i$ $f_x$ $f_y$ $f_z$ $x_1$ $y_1$ $z_1$ $x_2$ $y_2$ $z_2$ ...

First a cutoff is specified such that particle pairs with a distance greater than $C$ contribute zero to the force and energy calculation. If $C$ is negative, the cutoff is infinity. Each of the following lines begins with an energy $e$ followed by an index to an particle for which the force is computed, $f_i$. The first particle index is 1. This is followed by the Cartesian components of the force. Finally the Cartesian coordinates for each particle in the system are given. The equations that result from the optimization will be in terms of the independent variable $X$ which represents the interparticle distance for a pair as calculated from the supplied positions. Therefore, $X$ should be added to the primitive set as described for pair.

PAIR POTENTIAL USING ONLY THE FORCE (-t pairf)

This template is similar to paird with the difference that only the forces are used in fitness evaluation. This style can therefore allow for much faster optimization followed by refinement by switching to style paird. For this reason, the format for the input data file is identical to that for paird. The energies specified in this file are ignored.

PAIR POTENTIAL USING ONLY THE X-FORCE (-t pairf1)

This template is similar to pairf except that only the $x$ component of the force is utilized for fitness evaluation. The input file format is identical to that for pairf and paird.

TWO/THREE BODY POTENTIALS (-t twothree)

This template evaluates two summations for the potential energy and can be used to fit potentials that include a 2-body term and a 3-body term. The form for the expression is:

$$p = \frac{1}{n} \sum_i^n \phi(X_i) + \frac{1}{m} \sum_i^m \| (R_{11}, R_{21}, A_1)$$

The sample data therefore consists of a set variables $X_1, ..., X_n$ that are evaluated in the first summation and a second set $R_{11}, ..., R_{1m}, R_{21}, ..., R_{2m},$ and $A_1, ..., A_m$ that are evaluated in the second summation, where $n$ is not necessarily equal to $m$. For a 2/3-body potential, $X$ might represent the interparticle distances in the 2-body part of the potential. For the 3-body part, $R_1$ and $R_2$ might represent the distances from particle 1 to particles 2 and 3 and $A$ might represent the angle between the corresponding vectors. The format for the input data file is:

# Comments for the input file
e TWO $X_1$ ... $X_n$ THREE $R_{11}$ $R_{21}$ $A_1$ ... $R_{1m}$ $R_{2m}$ $A_m$

where $e$ is the energy of the configuration. In order to use this style, the variables $X$, $R_1$, $R_2$, and $A$ must be added to the primitive set:
The equations for $g$ and $h$ are stored in the same expression tree where $g$ is the left subtree of the root node and $h$ is the right subtree of the root node. For this template, the root node is meaningless.

**TWO/THREE BODY POTENTIALS WITH FORCE (-t twothree)**

This template evaluates the `twothree` potential style, but also evaluates the force for a single particle in each configuration in the fitness function. This is done in an identical manner to the pair potential style and the input format is also identical. Using the cutoff and particle positions, the vector $X$ is calculated to contain all particle pairwise distances smaller than the cutoff. Likewise, for all particle triplets, the vectors $R_1$, $R_2$ and $A$ are calculated to contain the distances between the center atom and the other two atoms and the angle between the corresponding vectors if the two distances are both smaller than the cutoff. As with the other `twothree` styles, the variables $X$, $R_1$, $R_2$, and $A$ should be added to the primitive set.

**PARALLEL PM-DREAMER**

PM-Dreamer can be run in parallel using an island model. In serial, PM-Dreamer uses the Open Beagle model allowing for multiple populations with individual movement according to migration operators. In parallel PM-Dreamer allows for multiple islands, 1 per process, to be run. Each island can contain multiple populations with migration controlled by the standard operators. Migration between the islands is controlled by additional operators which are described below. The output for each island is written separately to the files `gp_force_0.log`, `gp_force_1.log`, ... and `gp_force_0.obm(.gz)`, `gp_force_1.obm(.gz)`, ... It should be noted that in the current implementation, random seeds only produce the same output when run on the same number of processors. When running in parallel, the `MPITerminateOp` should be used to assure proper termination of all processes in a run. The additional operators available for parallel execution are:

**MigrationMPIOp**

Each time `ec.mig.mpi_interval` generations passes, `ec.mig.mpi_size` individuals from each population on an island migrate to a randomly chosen island and are replaced with immigrants from a second randomly chosen island. The random islands are chosen such that all islands will participate in migration at each iteration. The operator does not perform migration between populations on the same island. This can be achieved using standard migration operators in addition to `MigrationMPIOp`.

**HFCompMPIOp**

This implements a distributed parallel algorithm for the Hierarchical Fair Competition inspired by the work of Hu and Goodman. (Similar to the serial `HierarchicalFairCompetitionOp`). This operator should not be used with the serial `HierarchicalFairCompetitionOp` operator. In this algorithm, a fitness threshold is chosen such that any individuals from a population with index $i$ will migrate to the population $i+1$ if their fitness is better than the fitness threshold for that population $i+1$. If any populations has excess individuals following migration, the least
fit individuals are killed off. Random individuals are added to account for any shortages. This promotes a hierarchy of populations where the fitness of the best individuals improves with the population index. The migration occurs through all populations on a single island followed by migration of individuals of the last population of one island to the first population of another. In order to achieve parallel efficiency, there is a 1-step lag from the time individuals migrate out of an island to the time they arrive at the next. The fitness thresholds for the populations can be set in 2 ways. In the default, ec.hfc.first is set to -1 and the fitness threshold for a population is set to a value where the threshold is greater than ec.hfc.percentile of the population. For example, if ec.hfc.percentile is 0.85 the fitness threshold for a population is set to the value of the individual whose fitness is worse than only 15% of the population. In the case, the fitness thresholds are adaptive. In the second approach, the fitness thresholds are fixed. ec.hfc.first (float greater than 0 and less than 1) is set to the fitness threshold of the first population accepting incoming individuals. The thresholds for the subsequent populations are increased according to ec.hfc.scale (described below) to allow for thresholds up to but less than 1.0

**GP-StatsCalcFitSimpleMPIOp**

This operator can replace GP-StatsCalcFitnessSimpleOp to replace Vivarium statistics for a single island with Vivarium statistics for all processes in the log files. The hall-of-fame individuals are still reported per island however.

**MPITerminateOp**

This signals the application to terminate execution of all processes whenever a single island is terminated by any of the termination operators. Use this, for example, to ensure correct termination when a fitness of 1.0 has been found.

**TermMaxTimeOp**

Terminate after ec.term.maxtime minutes have passed. If compiled with MPI, this is the MPI wall time. Otherwise, this is the time calculated using c_time clock(). If set to zero, the operator is ignored.

The registers available for parallel execution are:

**ec.mig.mpi_interval**

The number of generations that must pass before a migration between islands occurs.

**ec.mig.mpi_size**

The number of individuals that migrate from each population of each island.

**ec.term.maxtime**

Terminate the evolution after this many minutes (default 60).

**ec.hfc.percentile**

Percentile of fitness measure to use as HFC migration threshold of next population. For example, a threshold of 0.85 means that the fitness used as threshold to accept migrant into following population is taken as the fitness of the individual that is
better than 85% of the other individuals in its population. Default is 0.85. This value is ignored if ec.hfc.first is positive

**ec.hfc.first**

If negative, adaptive thresholds are used for HFC according to ec.hfc.percentile. If positive, the register must be greater than 0 and less than 1.0. The thresholds for the populations in HFC are then set evenly spaced fixed values between first and 1.0.

**ec.hfc.scale**

This parameter is used to adjust how the fitness thresholds of populations are scaled if adaptive thresholds are not used (ec.hfc.first>0). The ratio between the fitness thresholds of populations is given by ec.hfc.scale to create a geometric series between ec.hfc.first and 1.0. If ec.hfc.first is 1, the fitness thresholds are evenly spaced. If ec.hfc.first is >1, more of the fitness thresholds are at lower fitnesses. If <1, more are at higher fitnesses. The default value is 1.

**ec.hfc.interval**

Interval between each hierarchical fair competition migration, in number of generations. An interval of 0 disables HFC migrations. Default is 1.

**HYBRID PM-DREAMER**

PM-Dreamer supports hybrid optimization of functional forms, allowing for local optimization of constants in the expression tree. This is accomplished by adding the GP-HybridOptOp operator. The registers that parameterize the operator include:

**gp.hybopt.indpb**

The frequency with which hybrid optimization is performed on an individual. The default value is 0.05.

**gp.hybopt.primit**

The name for the constants in the tree that are optimized. Default is E.

**gp.hybopt.type**

The type of optimization to be performed. If the value is 0, all constants in an expression tree are optimized using multidimensional Nelder/Mead Simplex algorithm. If the value is nonzero, a random constant in the expression tree is optimized using 1D minimization (also with Simplex). The default value is 1.

**gp.hybopt.maxi**

The maximum number of iterations of local optimization to be performed. Default is 10.
RESTARTING OPTIMIZATIONS

Simulations can be restarted using the -c flag. This requires that the ReadRestartOp operator be present in the beagle_config_file. The ReadRestartOp replaces the MilestoneReadOp in OpenBeagle. When restarting, the individuals and the generation number are read. The data, template_style, and parameters from the previous run are not read in. This allows the user to continue a run with new data, template_style, and/or configuration parameters. When restarting a run with new data or a new template_style, the fitness of all individuals are recalculated and the Hall of Fame individuals are updated with any changes that result from the new fitness evaluation. Because the restart will start using the last generation from the milestone files, the ec.term.maxgen register may need to be increased to allow for a larger number of generations. When restarting in parallel, if a smaller number of processors is used, the individuals from the higher rank processes will be thrown out.

VECTORIZATION

PM-Dreamer allows for vectorized evaluation of expression trees using the -z flag. When vectorization is enabled, the expression tree for a given individual needs to be parsed only a single time using the vector(s) of values necessary for energy/force calculation. This provides an improvement in speed because it prevents multiple parsing of the same expression tree and the potential for SIMD compiler optimizations. The configuration file does not need to be changed to utilize vectorization; internal replacements of the standard primitives and fitness operators are performed to allow vector math operations to be performed. Although the runs with vectorization should produce identical results, changes due to finite precision and the order of summation operations can result in different results. Because certain Open Beagle primitives have the argument types hard-coded, vectorization is left as an option to aid in compatibility with future versions.

AUTHORS

W. Michael Brown
Appendix B - GP-Force Performance for Different Parameter Sets

Figure B-1. GP performance at varying tournament selection sizes for a Lennard-Jones test case consisting of 10 configurations with 55-65 pair-interactions per configuration. Each data point is the mean best fitness calculated as an average over 50 runs on 32 processors with 5000 individuals per processor. HFC was run using fixed fitness thresholds ranging from 0.11 to 1.0. HFC migration was performed every generation. For the other replacement strategies, random migration of 50 individuals was performed every 2 generations. For the random case, the population is destroyed at each generation and new random individuals are created. For HFC, the crossover probability was 0.9 and for all others it was 0.8. The probability of hybrid optimization was set to 0 in these cases; the probability of all other types of mutation was set at 0.05. The number of generations in the evolution for each case is shown below.
Figure B-2. GP performance at various crossover probabilities. Runs were performed as described in Figure B-1 using a tournament selection size of 6 for Generational, HFC, and Steady-State replacement strategies and 8 for the Simple replacement strategy.
Figure B-3. GP performance using various probabilities for constant mutation.
Figure B-4. GP performance using various standard mutation probabilities.
Figure B-5. GP performance using various shrink mutation probabilities.
Figure B-6. GP performance using various swap mutation probabilities.
Distribution

1    Alexander Slepoy, NNSA, Alexander.Slepoy@nnsa.doe.gov (electronic copy)
1    MS-1315   Mark J. Stevens, 8331 (electronic copy)
1    MS-1316   William M. Brown, 1412
1    MS-1316   Steven J. Plimpton, 1416 (electronic copy)
1    MS-1316   Mark D. Rintoul, 1412 (electronic copy)
1    MS-1318   Jean-Paul Watson, 1412
1    MS-1322   John B. Aidun, 1435 (electronic copy)
1    MS-1322   Paul S. Crozier, 1435 (electronic copy)
1    MS-1322   Peter A. Schultz, 1435
1    MS-1322   Aidan P. Thompson, 1435
1    MS-0123   D. Chavez, LDRD Office, 1011
1    MS-0899   Technical Library, 9536 (electronic copy)