Convergence of symbolic regression using metaheuristics

Ben Cardoen
ben.cardoen@student.uantwerpen.be

May 7, 2017

1 Abstract

Symbolic regression (SR) fits a symbolic expression to a set of expected values. Amongst its advantages over other techniques is the ability for a practitioner to interpret the resulting expression, determine important features by their usage in the expression, and an insight into the behavior of the resulting model (e.g. continuity, derivatives, extrema). SR combines a discrete combinatoric problem (combining base functions) with a continuous optimization problem (selecting and mutating constants). One of the main algorithms used in SR is Genetic Programming (GP). The convergence characteristics of SR using GP is still an open issue. The continuous aspect of the problem has traditionally been an issue in GP based symbolic regression. This paper will study convergence of a GP-SR implementation on selected use cases known for bad convergence. We introduce modifications to the classical mutation and crossover operators and observe their effects on convergence. The constant optimization problem is studied using a two phase approach. We apply a variation on constant folding in the GP algorithm and evaluate its effects. The hybridization of GP with 3 metaheuristics (Differential Evolution, Artificial Bee Colony, Particle Swarm Optimization) is studied.

2 Introduction

2.1 Overview

2.2 Symbolic Regression

2.2.1 Compared to other techniques

2.2.2 Interpreting results

2.2.3 Applications

2.3 Convergence

2.4 Genetic Programming

2.5 Constant optimization problem

3 Design

In this section we will detail the design of our tool, the algorithm and its parameters.

3.1 Algorithm

3.1.1 Input and output

The algorithm accepts a matrix X = n x k of input data, and a vector Y = 1 x k of expected data. It will evolve expressions that result, when evaluated on X, in an 1 x k vector Y' that approximates Y. N is the number of features, or parameters, the expression can use. K is the number of datapoints. The algorithm makes no assumptions on the domain or range of the expressions or data set. While domain specific knowledge can be of great value, in real world situations such data is not always known. We aim to make a tool that operates under this uncertainty.

3.1.3 Entities

We will describe briefly the important entities in our design. In Figure 2 the architecture of our tool is shown. Functionality such as IO, statistics and plotting is left out here for clarity. The interested reader is referred to the source code documentation.

Algorithm The algorithm hierarchy extends each instance with new behavior, only overriding those functions needed. Code reuse is maximized here by the usage of hook functions, which in the superclasses result in a noop operation. An example of this is the evolve function, by and large the most complex function of the algorithm. By using hooks such as *requireMutation()* this function can remain in the superclass, and the subclass need only implement the hook.

3.1.2 Control flow

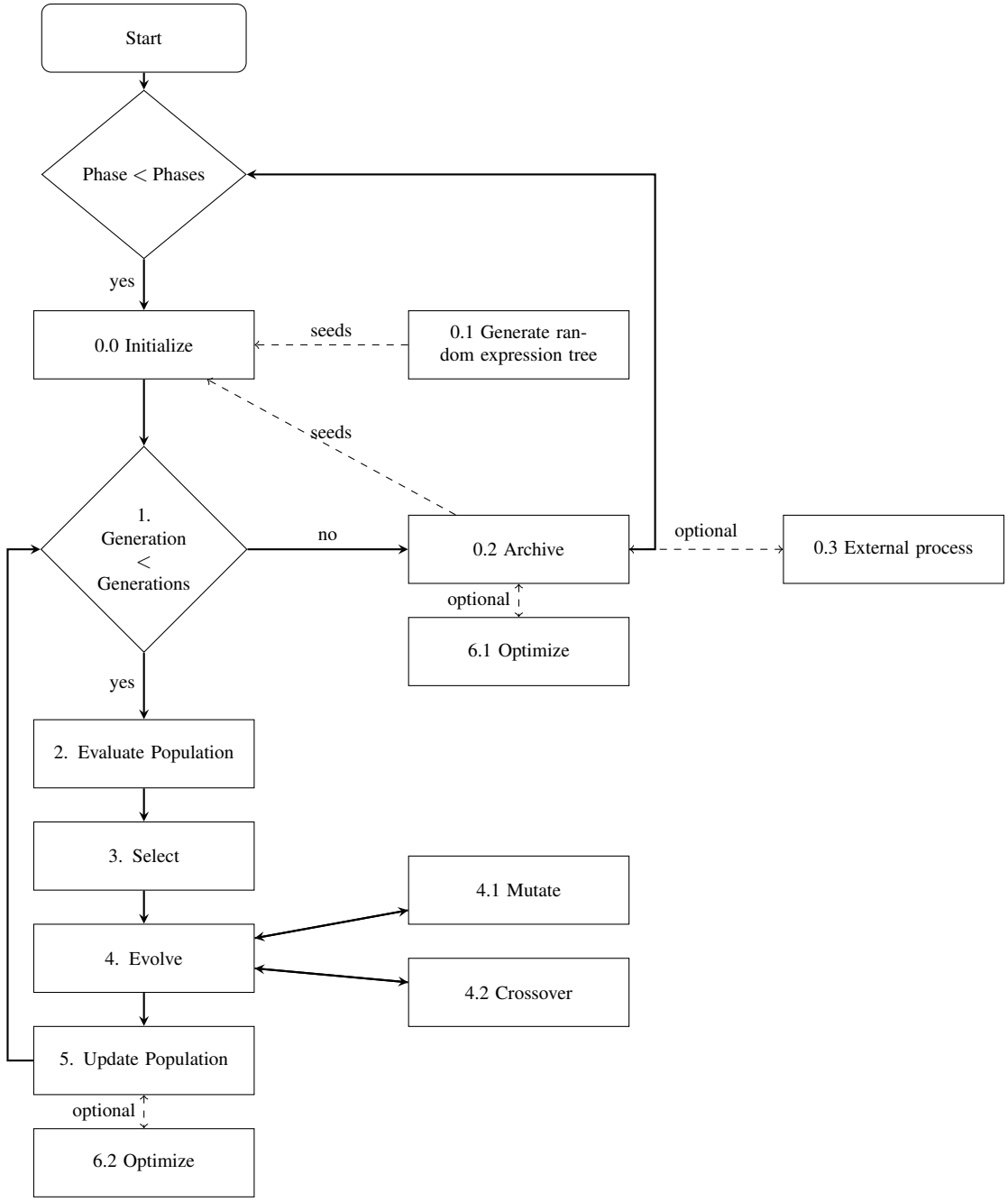


Figure 1: CSRM control flow.

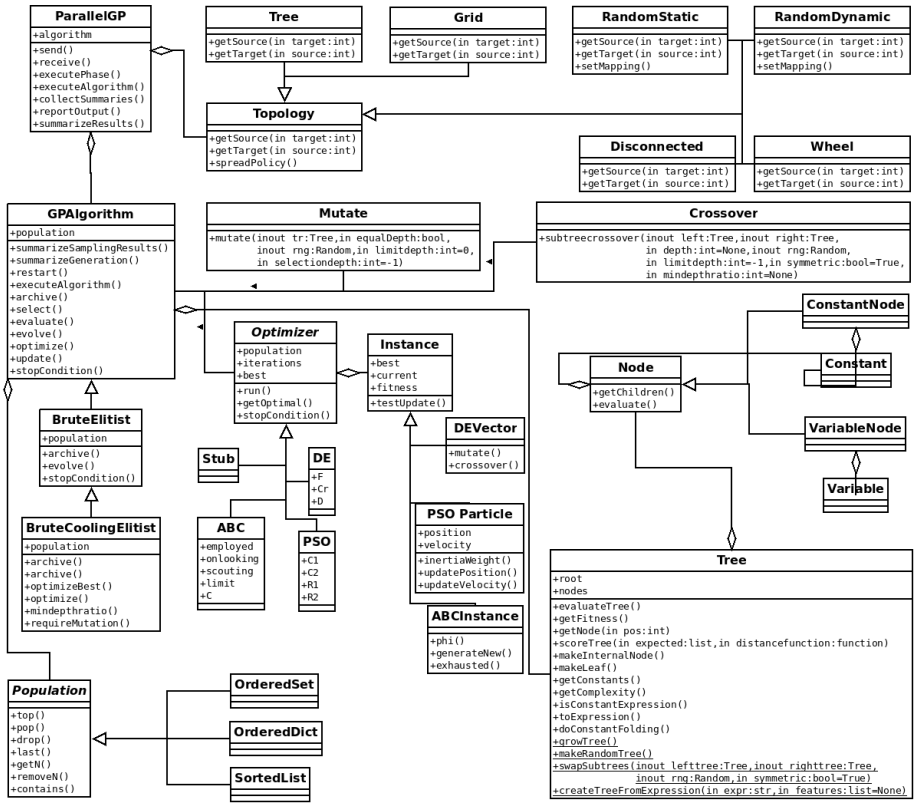


Figure 2: UML of CSRM.

Tree The Tree and Node classes represent the main data structure of CSRM's population. The Tree is comprised of Node objects, each of which holds an optional constant weight. Leaves are either features (VariableNode) or constants. In 3.8 we go in depth into the design choices made for this representation. The Tree class holds a few utility functions, for example to create trees from an expression using a parser in the tool module. Functionality needed by the mutation and crossover operators is present here as well. Of note are both construction functions, which allow for efficient generation of random trees. In section 3.3.1 we will cover the problem these functions solve.

Optimizer CSRM provides an interface to continuous optimizers. In section 4 we will go deeper into the specific algorithms used. The population data structure is shared between the subclasses. The instances in the population can be subclassed if specific functionality is needed by an optimizer. This approach allows us to reuse a large part of the code between algorithms. For convenience a stub optimizer is added. The optimization algorithms share configuration parameters, and encapsulate those that are distinct to their specific nature.

3.1.4 Implementation

Our tool is implemented in Python. The language offers portability, access to rich libraries and fast development cycles. The disadvantages are speed and memory usage compared with compiled languages (e.g. C++) or newer scripting languages (e.g Julia). Furthermore, Python's usage of a global interpreter lock makes shared memory parallelism not feasible. Distributed programming is possible using MPI.

3.2 Fitness

In this work we interpret optimization as a fitness minimization process. In the following we will discuss the fitness function and its behavior.

3.2.1 Distance function

The goal of the algorithm is to find f' such that

$$Y = f(X)$$

$$Y' = f'(X)$$

$$dist(Y, Y') = e$$

results in e minimal.
Not all distance functions are equally well suited for this purpose. A simple root mean squared error (RMSE) function has the issue of scale, the range of this function is [0, $+\infty$), which makes comparison problematic, especially if we want to combine it with other objective functions. A simple linear weighted sum requires that all terms use the same scale. Normalization of RMSE is an option, however there is no single recommended approach to obtain this NRMSE.
In this work we use a distance function based on the Pearson Correlation Coefficient. Specifically, we define

$$dist_p(Y, Y') = 1 - |r|$$

with

$$r = \frac{\sum_{i=0}^{n} (y_i - E[Y]) * (y'_i - E[Y'])}{\sqrt{\sum_{j=0}^{n} (y_j - E[Y])^2 * \sum_{k=0}^{n} (y'_k - E[Y'])^2}}$$

R has a range of [-1, 1] where 1, -1 indicate linear and negative linear correlation and 0 no correlation. This function has a range [0,1] which facilitates comparison across domains and allows combining it with other objective functions. The function reflects the aim of the algorithm. We not only want to assign a good (i.e. minimal) fitness value to a model that has a minimal distance, we also want to consider linearity between Y an Y'.

3.2.2 Diversity

Diversity, the concept of maintaining semantically different specimens, is an important aspect in metaheuristics. Concepts such as niching and partitioning are often used to promote this behavior, amongst other reasons to prevent premature convergence or even to promote multimodality. Our tool uses a simple measure that mimics the more advanced techniques stated above. It should be clear that for any combination of input and output data, there are a huge set of expressions with an identical fitness value. Such expressions can lead to premature convergence, consider a population of 5 where the 3 best samples all have fitness scores of 0.134. Our tool will aim to prevent retaining expressions that have identical fitness values. In contrast, in some metaheuristics [21] allowing replacement of solutions with identical fitness values (not duplication, but replacement), can actually help avoiding local minima.

3.2.3 Predictive behavior

The algorithm evaluates expressions based on training data X_t. X_t is an n x rk matrix based on the original input matrix X (n x k). R is the sampling ratio, the ratio between training and test data. After completion of the algorithm the population is ordered based on minimized fitness values calculated on the training data. In real world applications practitioners are also interested in the predictive qualities of the generates expression. In other words, how well do the expressions score on unknown data. In the final evaluation we score each expression on the full data to obtain this measure. While this gives us some information on how good an expression is on the full data set, we are also interested in how the convergence to this value progresses. If we add 10 more generations, or increase the population by a factor 1.5, what do we gain or lose in predictive quality of the expressions? To define this we use a correlation measure between the fitness values using the training data and those of the full data. This measure quantifies the predictive value of the final results. Finally, we calculate a correlation trend between the training fitness values at the end of each phase, and the final fitness values calculated on the full data. This trend describes the convergence process of the algorithm over time, specifically directed at the predictive value of the solutions found. It is important to note that the entire final population is considered in these calculations, not only the best. While ideal, there is no guarantee that the expression that has the lowest fitness value on the training data will score best on the full data set. We would like these measures to give us a descriptive capability for the entire process.

3.2.4 Convergence limit

As a stopcondition our tool uses a preset number of iterations. The user specifies the number of generations (g) and phases (r), and the algorithm will at most execute g x r iterations. Convergence stalling is implemented by keeping track of the number of successful operations (mutation or crossover). If this value crosses a threshold value convergence has stalled and the algorithm is terminated.

3.3 Initialization

Initialization is done using the 'full' method [17]. The algorithm has a parameter initial depth, each new expression in the population is created using that depth. Unless the maximum depth is equal to the initial depth, the algorithm will quickly vary in depth, evolving an optimal depth.

3.3.1 Invalid expressions

Generating a random expression is done by generating random subtrees and merging them. An important observation here is that randomly generated expressions can be invalid for their domain. The ubiquitous example here is division by zero. Several approaches to solve this problem exist. One can define 'safe' variants of the functions, in case of division by zero, returning a predefined value that will still result in a valid tree. The downside to this approach is that the division function's semantics is altered, a practitioner, given a resulting expression, would have to know that some functions are no longer corresponding entirely to their mathematical equivalents, and what 'safe' values the implementation uses. The other option is assigning maximum fitness to an invalid expression. While simple, this approach needs a careful implementation. From a practical standpoint wrapping functions in exception handling code will quickly deteriorate performance. This last problem is solved by a quick domain check for each calculation, avoiding exceptions.

Invalidity probability We define the probability that a randomly generated tree of depth d, with n possible features, k possible base functions, and j data points as q. With more complex problems d will have to be increased. GP is also susceptible to bloat [8], increasing d even further. This issue will affect generation of subtrees in mutation. With more datapoints the probability of at least one leading to an invalid evaluation will increase. An increase in d will lead to an exponential increase in the number of nodes in the tree. A node can be either a basefunction or a leaf (parameter or constant). For each additional node the probability q increases. We can conclude that q, while irrelevant for small problems and depths, becomes a major constraint for larger problem statements.

Bottom up versus top down There are two methods to generate a tree based expression : bottom up and top down. The top down approach is conceptually simpler, we select a random node and add randomly selected child nodes until the depth constraint in satisfied. The problem with this approach is that the expression can only be evaluated at completion, early detection of an invalid subtree is impossible. In contrast in a bottom up construction we generate small subtrees, evaluate them and if and only if valid merge them into a larger tree. This allows for early detection of invalid expressions. A downside of this approach is the repeated evaluation of the subtrees, which can be mitigated by caching the result of the subtree.

Disallowing invalid expressions in initialization We can generate random expressions and let the evolution stage of the algorithm filter them out based on their fitness

value, or we can enforce the constraint that no invalid expressions are introduced. The last option is computationally more expensive at first sight, since the algorithm is capable by definition of eliminating unfit expressions from the population. This can lead to unwanted behavior in the algorithm itself. For high q values we can have a significant part of the population that is at any one time invalid. This can lead to premature convergence, similar to a scenario where the population is artificially small or dominated by a set of highly fit individuals. Another observation to make is that the algorithm will waste operations (mutation, crossover) on expressions that are improbable to contribute to a good solution. While more expensive computationally, we therefore prohibit generation of invalid expressions in the initialization.

3.4 Evaluation and cost

Evaluating a tree requires a traversal, once for each datapoint. In order to compare optimization algorithms across implementations practitioners can use as a measure the number of evaluations required to reach a certain fitness level. If this measure is used, one should take into account that not all evaluations are equal. With the trees varying in depth and density the evaluation cost varies significantly. Furthermore, evaluating "1 + 2" is computationally far less expensive than "log(3, 4)". A simple count of evaluation functions executed does not really reflect the true computation cost, especially when we consider the variable depth of the trees. Increasing the depth leads to an exponential increase in the number of nodes, and thus in the evaluation cost. Our tool uses a complexity measure that takes into account the density of the tree and which functions are used. A tree comprised of complex functions will score higher in cost than a corresponding tree using simple multiplications and additions. Although this is an option, we do not use this measure in the objective function. We would like to observe the effect of the cost, but not directly influence the algorithm. There is no direct link between more complex functions and an optimal solution. In certain domains the argument can be made that more complex functions are more likely to lead to overfitting, or more likely to lead to invalid trees due to smaller domain.

3.5 Evolution

In the evolution stage we apply two operators on (a selection of) the population. The operators are configured to constrain the depth of the modified trees, enforcing the maximum depth parameter. We trace the application of the operators during the execution using an effectiveness measure. Each time an operator application is able to lower the fitness of a tree, this measure increases. Using this measure we can gain insight when and how certain optimizations and modifications work inside the algorithm instead of simply observing the algorithm as a black box.

3.5.1 Mutation

Mutation works by replacing a randomly chosen subexpression with a newly generated. In our implementation this means selecting a node in the tree and replacing it with a new subtree. Our mutation operator can be configured to replace the subtree

with one of equal depth or a randomly chosen depth. The insertion point can be made depth-sensitive. A shallow node, a node with a low depth, is the root of a subtree with significant depth. Replacing such a subtree is therefore more likely to have a significant effect on the fitness value. Replacing a deep node will have on average a smaller effect. We will investigate this assumption in our experiments. In our implementation the mutation operator is applied using a cooling schedule, or on the entire population. The cooling schedule uses the current fitness and the current generation of a tree to decide if the tree is likely to benefit from mutation. This is similar to the approach in simulated annealing. Mutation introduces new information into the optimization process (a new subtree). This process can be constructive (lowering fitness) or destructive (increasing fitness). The idea behind the cooling schedule is that for fitter trees the mutation operation will not be able to improve fitness (decrease it), while for less fit trees it has a higher probability. This probability is estimated using the current generation and fitness value (relative to the population), and using this information a random choice is made whether or not to apply mutation. In the experiments section we will investigate if this assumption holds, and if it leads to gains in fitness and or effectiveness. The mutation operates has to generate new subtrees, and therefore the same issues seen in the initialization process which we discussed in section 3.3.1 apply here as well. The mutation operator will generate subtrees until a valid one is found. If the depth sensitive operation proves to generate equal or better fitness values, this could significantly reduce the computation cost of the operator.

3.5.2 Crossover

Crossover operates on 2 trees in the population. It selects subtrees from both, and swaps them between each other. The resulting set of 4 trees is scored and the 2 fittest replace the original trees. As with mutation crossover can be configured to operate on a set depth, or pick a random depth. It can also work symmetric, picking an equal depth insertion point in both trees. Crossover in our implementation can be applied in sequence to the entire population, with pairwise mating in order of fitness. Alternatively a random selection can be used. A variant of both, alternating between the two schedules based on a random choice is a third option. This is similar to the roulette wheel selection method, although without replacement. Crossover, unlike mutation, does not introduce new information in the sense that no new subtrees are generated. Crossover can be configured to work with a decreasing depth, operating only on deeper nodes at the later stages of the algorithm. The assumption for this mode of operation is similar to that made for mutation.

3.6 Selection

After evolution a decision has to be made on which expressions will be retained in the population. In our tool the population is fixed, so a replacement is in order. We use an elitist approach. If an expression has a lower (better) fitness value after mutation, it is replaced. In crossover we combine two expressions r, and t, resulting in two offspring s, u. From these four expressions the two with minimal fitness survive to the next generation.

3.7 Archiving

The algorithm holds an archive that functions as memory for best solutions obtained from the best expressions at the end of a phase., from seeding, or from other processes. At the end of a phase we store the j best expressions out of the population. J is a parameter ranging from 1 to the population size n. With j == n we risk premature convergence, with j == 1 the risks exists that we lose good expressions from phases which will have to be rediscovered. While there are numerous archiving strategies described in literature, we use a simple elitist approach. This means that there is no guarantee that the best j samples of phase i are retained, if they have fitness values lower than those present in the archive and the archive has no more empty slots, they will be ignored. This leads us to the size of the archive. While no exact optimal value for this exists, in order to function as memory between phases it should be similar in size to the amount of phases. The j parameter will influence this choice as well.

3.8 Representation and data structures

3.8.1 Expression

We use a tree representation for an arithmetic expression, where each internal node represent a base function (unary or binary), and each leaf either a feature or a constant.

Tree representation We use a hybrid representation of a tree structure. The tree is a set of nodes, where each node holds a list of children nodes. This representation favors recursive traversal algorithms. In addition to this representation, the nodes of a tree are stored in a dictionary keyed on the position of a node. This allows for O(1) access needed by the mutation and crossover operators. The overhead of this extra representation is minimal, since the keys are integers and only a reference is stored. A list representation would be faster in (indexed) access, but would waste memory for sparse trees. With a mix of unary and binary operators the average nodecount of a tree with depth d is $\frac{2^{d+1}-1+d}{2}$ = O(2^d) resulting in savings on the order of 2^d, with d the depth of the tree. The algorithm has the choice which mode of access to use depending on the usage pattern. As an example, selecting a random node in the tree is O(1), selecting a node with a given depth is equally O(1). Splicing subtrees is equally an O(1) operation.

Base functions The set of base functions is determined by the problem domain. For symbolic regression we use the following set:
+, -, %, /, max, min, abs, tanh, sin, cos, tan, log, exp, power
A problem with this set is the varying domain of each, why may or may not correspond with the domain of the features. A possible extension is to use orthogonal base functions such as Chebyshev polynomials. In our solution the functions listed correspond with their mathematical equivalent, e.g. division by zero is undefined. Other constraints are based on floating point representation (e.g overflow).

Constants Constants are fixed values in leaves in the tree. The representation also allows for multiplicative weights for each node, which can be used to optimize the final solution. In contrast to the base functions with a limited set to choose from, constants have the entire floating point range at their disposal. The probability of selecting a 'right' value is extremely small, and domain information is lacking for these constants, it depends on the entire subtree holding the constant. We select a constant from a small range and allow the algorithm to recombine these constants later to larger values. This limiting of the search space can lead to the algorithm converging faster to a suboptimal solution. The constants are reference objects in the tree, and so can be accessed and modified in place by an optimizer.

Features Features are represented as an object holding a reference to a set of input values, and always occur as leaves. The choice for a random leaf is evenly distributed between constants and features.

3.8.2 Population

The population of trees is kept in a sorted set, where the key is the fitness value and the value is a reference to the tree representing the expression. Sorted datastructures are notably lacking from Python, we use the sortedcontainers [9] module which provides sorted datastructures that are designed with performance in mind. This representation allows for fast access in selecting a (sub)set of fittest trees. It also allows for an O(1) check if we already have an equivalent solution, a tree with a fitness score which we already have in the population. This allows our diversity approach mentioned in 3.2.2. In Figure 2 we see that the actual data structure used as population is interchangeable. If duplicate fitness values are allowed, and membership testing is no longer needed, a sorted list could be used instead.

3.9 Parameters

In this section we briefly list the main parameters of the algorithm and their effects on convergence, runtime and complexity.

3.9.1 Depth

The depth of trees van vary between an initial and maximum value. If we know in advance an optimal solution uses 13 features, the tree should have at least 13 leaves in order to use each feature at least once. This requires a depth of at least 4. In practice the depth will need to be greater due to the use of unary functions, and bloat. Bloat is a known issue in GP [8] where the algorithm, unless constrained by limits or an objective function that penalizes depth, will tend to evolve deep trees without gaining much in fitness. In the worst case entire subtrees can be evolved that do not contribute to the fitness value, mimicking introns in genetics. These can still have a valid purpose, serving as genetic memory. Their disadvantage is also clear: a computational overhead without clear effect on the fitness value. A similar problem arises with the generation of constants. Suppose we would like to generate the constant 3.14, the probability

of generating this by a single random call is extremely small. The algorithm will try to build an expression fitting 3.14, for example ”1+(3*1) + 28/200”. The problem with this is that this process is highly inefficient. It uses iterations that, if a more efficient approach exists, could be used to optimize the fitness value of the tree further. The constant expression wastes nodes that could be used to improve the tree. One approach is folding such expressions into a single constant, which mitigates some of the effects mentioned but does not prevent such subtrees from forming. Without a solution for this issue, the user would have to take this into account and increase the depth parameter. From 3.3.1 we know that the initialization process and the mutation operator will become more costly exponentially with the increase in depth. A similar argument can be made for the time and space complexity of operating on deeper trees, both increase exponentially.

3.9.2 Population size

The population size is directly related to convergence speed. The catch is the quality of the solution, a very small population will lead to premature convergence, the population is lacking in diversity (or information viewed from a different perspective). A large population on the other leads to a large increase in runtime, unless the operators only work on a subset of the population. The optimal value is problem specific, but values in the range of 20-50 give a good balance for our implementation.

3.9.3 Phases and generations

The execution of the algorithms comprises of g generations and r phases, resulting in a maximum of g x r iterations. For both parameters a too small value will hinder convergence, while a high value can lead to overfitting. The optimal value is domain specific and dependent on the iterations required to approximate the expected data. This is by virtue of the problem statement unknown. There is a subtle difference between these parameters. Each phase a reseed of the algorithm is done using the archive. This archive holds the best results from previous phases, external seeds and in a distributed setting the best best solutions from other processes. The remainder of the population is initialized with random tree. These trees introduce new information into the optimization process, and while expensive and with a low probability of improving fitness, nonetheless will help avoid premature convergence. If the archive size is less than the population size, new trees will always be introduced. The rate at which the archive fills is dependent on the number of phases and a parameter which determines how many trees are archived. In a distributed setting this process accelerates using the input of other processes. If the archive is full after i phases, and the archive size is equal to the population, further phases will no longer have the benefit of newly generated trees. This does not prevent convergence, but could reduce the convergence rate in some scenarios. Increasing g and r both can lead to overfitting, but in order to decide on r we also have to take into account the archiving strategy. In order to find a good starting value for g one can look at the population size. Diffusion, where the information from each expression is shared with the others, will require at least the same amount of generations as there are expressions, depending on the operators and the individual fitness

of the expressions. Concentration, where we only look at maximizing the few existing fittest expressions, requires far less generations, but risks premature convergence.

3.9.4 Samples

The user can provide the algorithm with input data, or can specify a range from which to sample the input data. In addition, the ratio between training and testing data can be adjusted. Care should be taken in tuning this value. A low ratio will increase the probability that evolved solutions are invalid on the test data, while a high ratio will lead to overfitting.

3.9.5 Domain

Domain specific knowledge can significantly reduce the time needed to converge to a solution. In our implementation we assume no domain knowledge. While this increases the complexity of obtaining a good solution, it also makes for a fairer evaluation of the algorithm and optimizations used.

3.10 Incremental support

Our tool supports incremental operation. The user can provide seeds, expressions that are known or assumed to be good initial approximations. The algorithm writes out the best solutions obtained in an identical format. By combining this the user can create a Design Of Experiment (DOE) setup using the algorithm. As a use case, suppose the user wants to apply symbolic regression on the output of an expensive simulation. The simulator has k features or parameters, each with a different domain and j datapoints. The user wants insights into the correlation of the parameters. A naive approach would be to generate output data for a full factorial experiment, and feed this into the CSRM tool. For both the simulator and the SR tool the cost of this approach would be prohibitive. It is likely that some features are even irrelevant, leading to unnecessary computation. Instead we can opt for a combined DOE approach. We start with a subset of features k' $<$ k and datapoints j' $<$ k. The simulation results Q of this subset are then given to the CSRM algorithm. It generates a solution, optimized for this subset of the problem. The user then adds more parameters, k' $<$ k" $<$ k and/or datapoints j' $<$ j" $<$ j, runs the simulator again. The resulting output is given the CSRM tool, with Q as seed. This seed is used as memory of the previous run on the smaller input set. Unless there is no correlation between the incrementing datasets the CSRM tool can use the knowledge gained from the previous run to obtain faster convergence for this new dataset. In addition, by inspecting Q the user can already analyze a (partial) result regarding the initial parameter set. Suppose k = 5, and only 3 parameters are used in Q. Then the user can exclude the missing two parameters from the remainder of the experiment, as these are unlikely to contribute to the output. By chaining the simulator and CSRM tool together in such a way, an efficient DOE approach can potentially save both simulation and regression time, or result in increased quality of solutions. The advantages are clear :

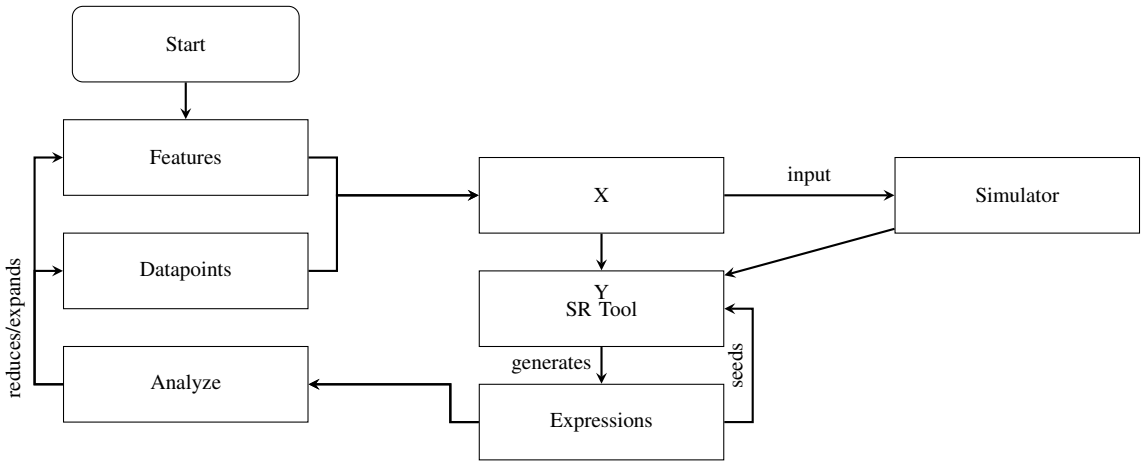


Figure 3: Incremental DOE using CSRM and a simulator.

- The SR tool can start in parallel with the simulator, instead of having to wait until the simulator has completed all configurations.
- Reducing irrelevant parameters detected by the SR tool can save in configuration size for the simulator.
- The SR tool can reuse previous results as seeds, and tackles and slowly increasing search space with those initial values instead of starting without knowledge in a far larger search space.

Possible disadvantages are :

- The SR tool is not guaranteed to return the optimal solution, it is possible the process is misguided by suboptimal solutions. This risk exist as well in the full approach.
- Interpretation of the results can be needed in order to make the decision to prune features.

We will investigate this approach in our use case in section 5. The following diagram illustrates a DOE hypercube design using our tool and a simulator.

3.11 Statistics and visualization

Stochastic algorithms are notoriously hard to debug. By virtue of the problem statement we do not know whether the returned solution is what is expected. The size of the search space makes detecting all edge cases infeasible. In addition the algorithm functions as a black box, where output is presented to the user without a clear trace indicating how or why this output was obtained. Both for developer and practitioner insight into the algorithms inner workings is vital. Our tool provides a wealth of

runtime statistics ranging from fitness values per expression for each point in the execution, convergence behavior over time, depth and complexity of the expressions, cost of evaluations, effectiveness of operators, correlation between training and test fitness values and more. These statistics can be saved to disk for later analysis, or displayed in plots in a browser. Using this information the user can tune the parameters of the algorithm (e.g. overfitting due to an excessively large number of phases), the developer can look at how effective new modifications are (e.g. new mutation operator) and so on. It is even possible to trace the entire run of the algorithm step by step by saving the expressions in tree form, displayed in an SVG image rendered by Graphviz [7]. In Figure 4 a selection of the collected statistics on a testfunction is shown. The third of our set of testproblems was used with a depth $\in$ [4,10], 30 generations, populationsize 30, and 4 phases.

3.12 Conclusion

In this section we have covered in detail the design of the CSRM tool, highlighting the choices made.

4 Constant optimization

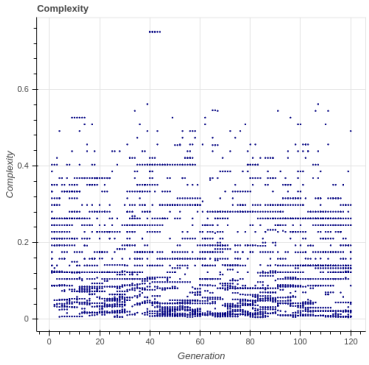
4.1 Constant Optimization

Selecting base functions is a combinatoric, discrete problem. Selecting the right constants to use is a continuous problem, that GP tries to solve with a combinatoric approach. There are several aspects to this issue, we will go over each individually and show our approach.

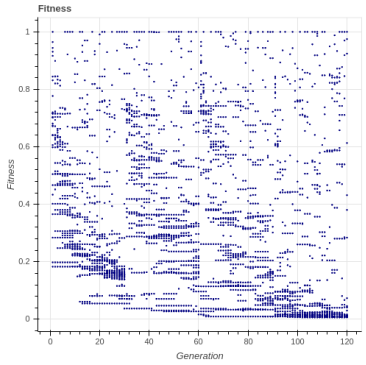
4.1.1 Initialization revisited

During initialization it is possible that a generated tree represents a constant expression. Such an expression is only a valid approximation if no feature has an influence on Y, which is an unlikely edge case. A constant expression is of no use to the algorithm, but without measures to prevent or remove these they will still be formed. Ideally such an expression will have a worse fitness value than non constant expressions, and will be eventually filtered out. Since detecting a constant expression is feasible in worst case O(n) with n the number of nodes in the tree, a more efficient approach is preventing constant expressions from being generated.

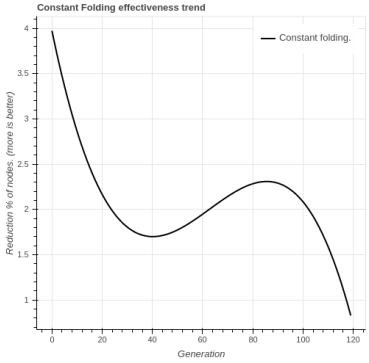
Constant expression detection An expression is a constant expression if all its children are constant expressions. As a base case, a leaf node is a constant expression if it is not a feature. This problem statement allows us to define a recursive algorithm to detect constant expressions. It should be noted that its complexity is O(n) only in the worst case, when the tree is a constant expression. Upon detecting a non constant subtree, the algorithm returns early without a full traversal.



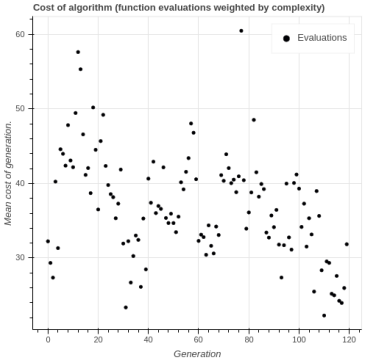
(a) Scaled complexity over generations.



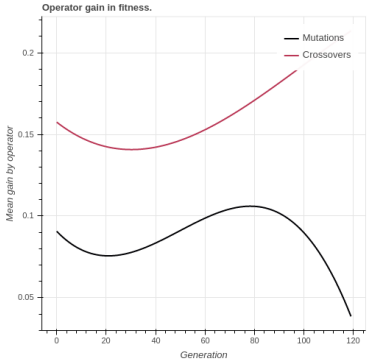
(b) Fitness values over generations.



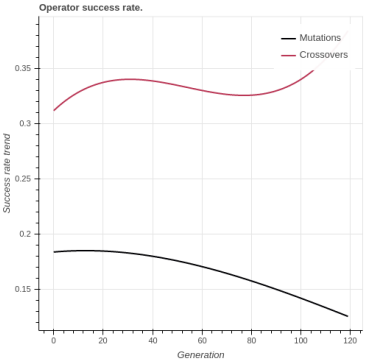
(c) Constant folding savings.



(d) Mean evaluation cost.



(e) Operator gain.



(f) Operator success rate.

Figure 4: Selection of visualizations generated by CSRM.

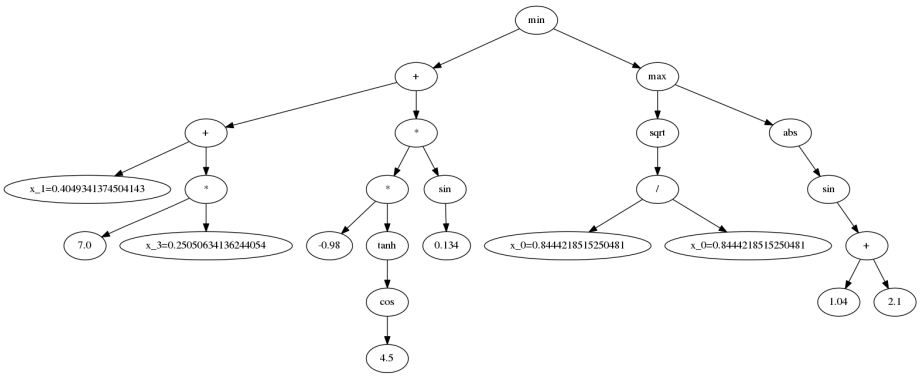


Figure 5: Tree before subtree folding.

Preventing constant expressions Using the checking procedure in the initialization, a tree marked as a constant expression is not allowed in the initialization procedure. It is still possible to create constant expressions by applying mutation and crossover. If the left subtree of a node is a constant expression, and the right is not, and this right is replaced by either mutation or crossover with a constant expression then the tree becomes a constant expression. The mutation operator will not generate constant subtrees, so this leaves only crossover. Our tool does not prevent constant expressions from forming in this way, the evaluation following crossover will filter out the constant expressions using evolutionary pressure.

4.1.2 Folding

Constant subtree problem A tree can contain subtrees that represent constant expressions. This is an immediate effect of the GP algorithm trying to evolve the correct constant. This can lead to large subtrees that can be represented by a single node. Nodes used in such a constant subtree are not available for base functions. They waste memory and evaluation cost, without their presence the tree could become a fitter instance. We will evaluate the size of these effects in the experiments section.

Constant subtree folding It is possible to use a depth sensitive objective function to try to mitigate this effect, but a more direct approach is replacing the subtrees. Using the previous constant expression detection technique we can collect all constant subtrees from a tree. We evaluate each subtree, and replace it with the constant value it represents. This leads to savings in nodes and possibly a reduction in depth. These savings can have an effect on the convergence as well. Mutation and crossover will no longer operate on constant subtrees, and the iterations and place in the tree that becomes available can be used to improve the fitness of the tree. Constant folding requires an O(n) detection check, since the entire tree needs to be traversed. The folding operation itself is at worst O(n), if the entire tree is constant.

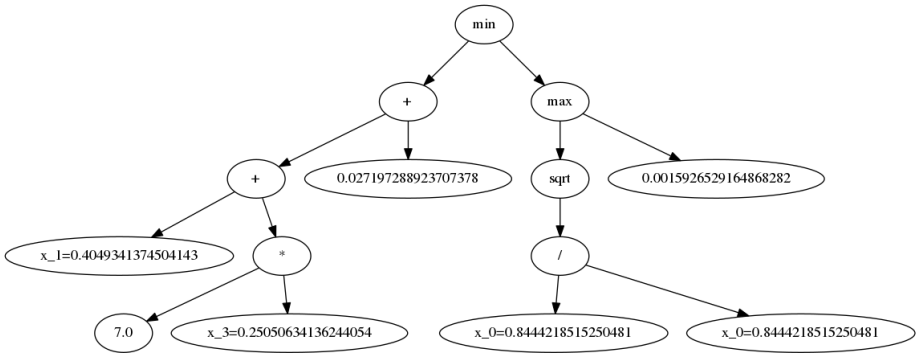


Figure 6: Tree after subtree folding.

Edge cases In Figure 6 we see the effect of applying the constant subtree folding. There are subtle edge cases that can't be detected using the above method. Consider the tree representing the expression

$$f(x) = \max(x, 4 + \sin(0.3))$$

The subexpression 4 + sin(0.3) is detected and folded into the constant 4.26. We should not stop there, since f can still be a constant expression if $\forall i : x_i < 4.26$. To verify this we need to evaluate f on all i datapoints. In contrast, the constant subtree detection code needs only 1 evaluation. In Figure 6 we see a similar case in the right subtree where the first value of x_0 the right subtree is indeed constant. Even if we should evaluate f for all i, this will not guarantee us that f is indeed a constant expression. All this check then proves is that for all i in the training data, f is constant. The testing data holds new values, for which f may or may not be constant. We conclude that this edge case cannot be prevented from occurring. Another case occurs in expressions of the form

$$f(x, y) = \tan(y) * \sqrt{x/x}$$

In this reduced example the node $\sqrt{x/x}$ is simply $\sqrt{1}$. Outside of this example, detecting such cases is non trivial. There is a clear benefit to do so, despite their low occurrence: if the domain of x includes 0 this tree is never generated because it leads to division by zero in a single datapoint. Discarding this expression is however not needed, since $\sqrt{1}$ is a valid subexpression. One way to solve this is to use mathematical software to simplify the expressions, and then convert them back to tree representations. Deciding to apply this step should be based on the cost versus benefit and the frequency of such occurrences.

Conclusion Constant detection and folding mitigate some of the side effects of the constant optimization problem, they do not lead to a more efficient finding of the 'right' constant values. For this we need a continuous optimizer, an algorithm designed specifically to optimize real valued problems.

4.2 Optimizers

Hybridizing GP Using a real valued optimizer in combination with GP is a known solution [19, 6]. Which algorithm to combine is a difficult question. To our knowledge there is no comparison made between optimizaton algorithms to find out which is a better fit to combine with GP.

Problem statement Given a tree with k constant leaves, with all constant subtrees folded, we would like to find the most optimal values for those constants resulting in a better fitness. It is vital to perform the constant folding step before optimization takes place. Suppose a given tree has on average k constants, which after folding become j with j $<=$ k. Without folding the optimizer has to solve a k dimensional optimization problem, whereas after folding the problem is only j dimensional. The underlying approximation only has j dimensions, so this step is a necessity.

Initialization In most optimization problems the initial solution is not known, only the problem domain and an objective function. The problem we face here does have an initial solution, namely that generated by GP. Instead of choosing random points in the search space, we therefore opt by perturbing this initial solution. An immediate problem here is that the optimizer may simply converge on the initial solution. This risk can be high, given that GP already has evolved it as a suboptimal solution. The problem faced by the population initialization step 3.3.1 reappears here. We could pick random values in the search space, but these are likely to generate invalid trees. A balance between exploration and exploitation is once again called for.

Domain Each tree instance has a completely different domain for each constant. We cannot therefore guide the optimizer with domain specific knowledge. This also reflects in the choice of parameters for the optimizer, which will have to be suboptimal for specific problem instances, as we want them to work on the largest set of problems.

Comparison and configuration In order to keep the comparison between the algorithms fair they are each configured as similar as is possible. Since they are all population based, and given a maximum number of iterations, all three share these same parameter values. Each application of the algorithm has a significant cost in comparison with the main GP algorithm. In a single iteration with population p, the GP algorithm is likely to perform 2n evaluations (mutation and crossover). If we give the optimizer a population of m and q iterations, it will execute at most in the order of m x q evaluations. Based on optimal value for PSO [12] we use a default population of 50, and 50 iterations. This means the cost of the optimizer will quickly dominate that of the main GP algorithm, depending on when and on what we apply it. We can apply the optimizer on each expression each iteration, only on a selection of the best expressions at each iteration, or only at the end of phase on all, a selection, or only the best expressions. In our experiments we will show the effect of these choices on convergence.

4.2.1 ABC

Artificial Bee Colony [10] is a relative new nature inspired optimization algorithm. It is not limited to continuous optimization, and has even been used for symbolic regression itself [11]. One of its key advantages over other algorithms is a lower parameter count. Optimal values for an optimization algorithm have a large impact on its convergence behavior, so much so that other optimizers can be required to find the parameters of an optimizer. With a small parameter set finding optimal values becomes easier. Finding optimal values for these parameters is quite often related to the problem domain, and as we have seen each instance here will have a new domain. ABC is good at exploration (thanks to the scouting phase) though sometimes lacking in exploitation. To resolve this ABC can be combined with more exploitative algorithms [18].

Algorithm ABC is a population based algorithm, using 3 distinct phases per iteration. We will refrain from using the nature analogy in describing the algorithm as it can be confusing. The algorithm maintains a set of potential solutions and a population of particles. Each particle is either employed, onlooker, or scout. An employed particle perturbs a known solution, and if an improvement in fitness is obtained replaces the old solution. If this fails a preset number of iterations, the solution is discarded and a new one scouted. Scouting in this context is generating a new potential solution. After the employed phase the onlooking particles decide, based on a fitness weighted probability, which solutions should be further optimized. In contrast to the employed particles they swap out solutions each iterations, whereas an employed particle is linked to a single solution. Finally exhausted solutions are replaced by scouted solutions.

Initialization The algorithm initializes its solution set by perturbing it. Perturbation is done by multiplying each constant with a random number $\in [-1, 1]$. Using the modified constants the tree recalculates its fitness value and the global best. Each instance records its best solution. The original source, the expression tree that we wish to optimize, is retained as a solution, and not perturbed. This can lead to premature convergence if the algorithm is configured to focus too much on exploitation versus exploration.

Modification In the employed stage, a solution x at time i is modified by combining it with another randomly chosen solution y. The choice for y is made with replacement, and obviously y $\neq$ x. With k random $\in [0, |x|)$

$$\begin{cases} x_{ij} = x_{ij} & j \neq k \\ x_{ij} = y_{ij} & j = k \end{cases}$$

The employed particle tries to improve the current source. If the modification leads to an improved fitness value, the solution is updated. Note that only a strict improvement warrants an update, an equal fitness value will not lead to an update. The onlooker phase is executed next. Each onlooker is assigned a solution using roulette wheel selection. For the set of solutions S we calculate fitness weights using:

$$w_i = \frac{1}{1 + f_i} \forall i \in S$$

Optimization in CSRM is a fitness minimization process, which leads to the fraction. A list is built using a cumulative sum of these weights. From this a uniform random number picks a single value based on these weights. A smaller fitness value will have a relatively larger section of the list compared with a larger fitness value resulting in a fitness bias in selection. Once assigned to an onlooker, the same modification process used by the employed particle is applied. Since selection is done with replacement it is possible that a single highly fit value is modified more than twice in an iteration. After the onlooker phase the algorithm checks in the scouting phase if any sources are exhausted. Exhaustion indicates that a solution can't be improved upon for at least *limit* modifications. Depending on the fitness value this limit can be reached faster for more fit values. There are several edge cases to consider here. Clearly this approach is beneficial if the exhausted solution has a poor fitness value. Improvement was impossible, so replacement is likely to improve the overall quality of the solution set by introducing new information. There is however the risk that algorithm discards highly fit solutions that fail to be improved. Discarding such a solution is warranted if that solution is a local optimum, but by the very nature of the problem statement we cannot know this in advance. We could prohibit discarding the best solution, even if the improvement limit has been exceeded. This does not solve all edge cases. If ABC is used in a multimodal search, it is still possible that valid solutions are discarded. Compounding the problem is that an equality update is not used. In CSRM we implement a strict check, so it is possible that equivalent solutions are discarded. With s scouts and e exhausted solutions $min(s, e)$ solutions will be replaced with new solutions. These are generated using a Normal distribution. In most optimization problem good starting positions are not known and thus a random point is selected. In our problem we already have a reasonably good solution, so we use this initial value as the mean of the normal distribution and generate values within 2 standard deviations around that mean. A configurable scaling factor is introduced, in our test problems we use 20. This value is trade-off between exploration and an increasing probability for generating invalid solutions. The values chosen have a far greater range than those generated by the initialization process. We do not know the domain of our problem, nor do we have the computational resources to cover the entire floating point range. The initial value is already evolved as a fit solution to our problem. The probability that the true optimum is far beyond the range of our initial value is estimated as low, though we can never be sure of this. The scouted solution replaces the exhausted solution. In our problem statement this can lead to issues. There is no guarantee that the scouted solution is actually valid. As we have seen in the initialization problem 3.3.1 this probability can be quite high. It is possible that an increasing part of the solutions are invalid. These will still be chosen to contribute in the modification step. It is unlikely though not impossible that they contribute to the convergence. In the worst case with enough iterations and a very small domain there it is possible that the entire population is replaced with invalid solutions. In our implementation the values of the threshold regulating exhaustion are chosen such that this is unlikely to occur. One solution here is to apply the same generation loop used in the GP algorithm, which keeps generating solutions until a valid one is found. Given the already high cost in evaluations the optimizer introduces we have chosen not to use this here. This applies for all optimizers implemented in CSRM, not just ABC. Note that the same problem is present in the initialization step, although far

less severe given the small perturbation applied there.

Selection A solution is updated if the fitness value is improved. The new fitness weights for the roulette wheel selection are calculated and the global best is updated. Unlike PSO a solution is only updated if an improvement is measured. In contrast to DE, equal fitness values do not lead to an update. An equality update allows an optimization algorithm to cross zero gradient areas in the fitness landscape. The influence other solutions have on each other in ABC is not as great as in DE. From the modification stage we also observe that at most 2 dimensions per iteration per solution are modified. In PSO the entire position, in all dimensions, is updated. In DE this depends on the Cr parameter which we discuss in 4.2.3. This distinction can have a large impact on convergence. The balance sought here is influenced by the interdependence of the constants. The modifications made by the algorithm can be seen as a process trying to extract information about this dependency. Suppose we have k constants, of which 2 have a large effect on the result. Then it only makes sense to modify those 2 dimensions, modification in the others does not gain anything except noise. The problem becomes more difficult if those two are correlated. Modifying all dimensions will not guide our optimization process as clearly as modifying only those 2. Modifying only a single one of the 2, as in ABC, is too strict as we lose the information about the correlation. We do not know in advance if our problem instances are separable or not. Related to this discussion, PSO can be sensitive to a bias along the axes of the dimensions [20] with improvements suggested in recent work [2].

Cost With a solution set of n, m employed, j scouts, i onlookers and k iterations we now look at the evaluation cost. Initialization requires n evaluations. Each iteration we execute m evaluations in the employed phase, i evaluations in the onlooker phase and at most j evaluations in the scouting phase. With m, j and i all $<=$ to n we have per iteration at most 3n evaluations. With our configuration, listed below, this value will be at most 2n. This results in an evaluation complexity of n + k 2 n, or O(kn).

Configuration

- limit = 0.75 * onlookers / 2 : If a solution can't be improved after this many iterations, it is marked exhausted and will be scouted for a new value. This limit is scaled by the number of dimensions per instance.
- population = 50 : This is the solution set, or set of sources.
- onlookers = 25 : The number of onlookers, instances that will be assigned solutions to exploit based on fitness values. Setting this value to half that of the employed finds a balance between exploitation and evaluation cost.
- employed = 50 : Instances that try to improve an assigned solution. If we use a value lower than the solution set we have to define an assignment procedure, which would mimick the onlooker phase. We therefore set the employed count equal to the size of the solutions set.

- scouts = 25 : This is a maximum value, up to this number are used to scout after a solution is exhausted. A higher scouting value leads to more exploration, a lower value favors exploitation. More exploration would result in the initialization problem dominating the runtime cost of the optimizer.

This configuration is guided by the findings in [10].

4.2.2 PSO

Particle Swarm Optimization [12] is one of the oldest population based metaheuristics. It consists of n particles that share information with each other about the global best solution.

Algorithm

Initialization Each particle is assigned a n dimensional position in the search space. A particle's position is updated using its velocity. This last is influenced by information from the global best and the local best. The concept of inertia is used to prevent velocity explosion [3]. Each particle is given a random location at start. In our application we already have an (sub)optimal solution, the constant values in the tree instances have been evolved by the GP algorithm. Rather than pick random values, we perturb the existing solution. This is a difficult trade-off. If the perturbation is not large enough the optimizer will simply converge on the known solution. If we perturb too greatly the risk for invalid solutions increases, rendering a large selection of the population invalid. We can initialize the population with n perturbed solutions or with n-1 with the initial value remaining intact. The n-1 solution is useful to test the algorithm, ideally the swarm will converge on the known best value. When applying the optimizer the n-perturbation approach is used, minimizing the risk for premature convergence. CSRM multiplies each constant with a random value in [0,1]. Each particle is assigned a small but non-zero velocity. The reason for this is again avoiding premature convergence. Without this velocity all particles are immediately attracted to the first global best. While attraction to this value is desired, it should not dominate the population. The small value of the initial velocity once again reflects an empirical discovered balance between exploration and exploitation. Each particle is assigned an inertiaweight. This value is one approach to combat the velocity explosion problem, which we will cover in 4.2.2. Finally the global best is recorded.

Modification The algorithm updates all particles in sequence, then records the new global best. Let d be the dimension of the problem, or in our case the number of constants in the tree to optimize. The velocity v at iteration i of a particle is updated using:

$$v_{i+1j} = w_i * v_{ij} + C_1 * (p_{ij} - g_{ij}) * R_1 + C_2 * (p_{ij} - G_{ij}) * R_2 \forall j \in [0, d)$$

with

- v_i Current velocity

- p_i Current position (set of constant values)
- g_i Local best
- G_i Global best
- C_1 Constant weight influencing the effect the local best has on the velocity.
- C_2 Constant weight influencing the effect the global best has on the velocity.
- w_i Inertia weight simulating physical inertia.
- R_1 Random value perturbing the effect the local best has on the velocity.
- R_2 Random value perturbing the effect the global best has on the velocity.

Without the inertia weight PSO has issues with velocity explosion, the velocity has a tendency to increase to large values. This increases the distance between particles, but more importantly is far more likely to generate positions that are no longer inside the domain of one or more of the dimensions. Inertia weighting will dampen this effect. The position is updated using :

$$x_{i+1,j} = x_{ij} + v_{ij} \forall j \in [0, d)$$

The R_1 and R_2 parameters make the modification stochastic, they introduce perturbations in the calculation. These changes have the benefit that they can break non optimal behavior resulting from the deterministic calculation. If there is a suboptimal best value (local or global) that leads to a too strong attraction and thus forcing premature convergence, we can with a certain probability escape from such a value by perturbing the velocity calculation. The C constants determine how strong the effect is of respectively the local best and the global best. This reflects the balance between exploration and exploitation respectively, where a particle is influenced more by its own information or that of the swarm. After all particles are updated, the new global best is recorded for the next iteration.

Selection An interesting difference with other algorithms is that the position is always updated, whether it improves the fitness or not. The local best records the best known position, but the particle is allowed to visit positions with lower fitness values. This allows it to escape local optima. The global best is obviously only updated with an improved value. The comparison is strict, meaning that only a better value can become the new global best. This may not seem significant, but allowing equality updates can actually benefit an optimization process. To see this it helps to view the fitness domain as a landscape with troughs, valleys and heights. Allowing for equality updates allows the global best to move despite no apparent improvement in fitness. In this analogy we represent a (sub) optimal value with low points in the landscape. This allows it to cross areas where a zero gradient is observed, for example between two low points where the lower one is a local optima.

Cost With a population size of n, the algorithm requires n fitness evaluations per iteration. The computational cost of updating the velocity and position is small given the evaluation cost of an expression tree over several datapoints. However, it is linear in d, the number of dimensions. If the tree increases in size, the number of constants can increase in the worst case exponentially. On average due to the construction algorithm we expect that half the leaves in the tree are constants. Given our discussion of the constant folding algorithm we know that a full binary tree is unlikely, so the number of constant nodes is equally unlikely to increase exponentially, but will nonetheless scale poorly. In our implementation we will halt the algorithm if it cannot improve the global best after k/2 consecutive iterations, where k is the maximum number of iterations it can execute. The initialization stage adds another n evaluations, in addition to n per iteration. The total cost in fitness evaluations is therefore n(k+1), resulting in a worst case evaluation complexity of O(nk).

Configuration This overview gives the values of each parameter used in CSRM's PSO implementation.

- C_1 = 2
- C_2 = 2 : Setting both to 2 is recommended as the most generic approach [13].
- $w_i = \frac{1+r}{2}$ with r random in [0,1] : In early implementation the inertia weight was kept constant [5] There are a large number of strategies for an inertia weight. Dynamically decreasing inertia may improve convergence significantly. We opt for a random inertia weight as it has been shown [1] to lead to faster convergence. Since our use case requires fast convergence on very limited iterations, this strategy is clearly favored.
- R_1, R_2 r with r random in [0,1]
- population = 50 : PSO is not sensitive to populations larger than this value, providing a robust default value. [14]

CSRM's optimizer does not set constraints on the domain of each constant, it is different for each problem instance. Finding the domain of a constant in the expression tree requires a domain analysis of the expression tree. With features in the tree involved, finding the exact domain is infeasible, given that some of the datapoints are unknown. It is therefore possible that a particle obtains values outside the valid domain of one or more constants, resulting in an invalid expression tree. This will result in the particle temporarily no longer contributing to the search process.

4.2.3 DE

Differential Evolution is a vector based optimization algorithm, or rather as the name implies, it operates by computing the difference between particles.

Algorithm The algorithm has a population of n vectors, similar to the other algorithms it holds a linear set of values to optimize, one per dimension.

Initialization Similar to our approach in initialization PSO, we perturb a known (sub) optimal solution. A vector stores its current value, and the best value.

Modification Each iteration the algorithm processes all vectors. For each vector $\vec{v}$, three distinct randomly selected vectors are selected. From these 3 vectors a new 'mutated' vector is obtained:

$$\vec{v} = \vec{w} + F(\vec{y} - \vec{z})$$

With $\vec{w}$,$\vec{y}$,$\vec{z}$ randomly chosen and not equal to $\vec{v}$.
The selection occurs with replacement. Several selection schemes exists, and the size of the selection is equally configurable. From this step the algorithm lends its name. The F factor influences the effect of the difference. Then we apply a crossover operation, using vectors x and v and probability parameter Cr. We select a random index j with $j \in [0, |x|)$. We then create a new vector u:

$$u_i = k_i \forall i \in [0, |x|$$

and k_i equal to

$$\begin{cases} v_i & i = j \vee r < Cr \\ x_i & i \neq j \wedge r \geq Cr \end{cases}$$

This is binomial crossover, another frequently used selection operation is exponential crossover.

Selection For a given selected vector $\vec{v}$ and created vector $\vec{u}$ we now test if $\vec{v}$ is a better candidate than $\vec{u}$, in other words has a lower or equal fitness value. Note the distinction here with PSO, the equality test allows DE vectors to cross areas without a gradient. If f($\vec{u}$) $\leq$ $\vec{v}$ the vector is replaced with $\vec{u}$. The global and local best are updated as well. A difference with PSO is that a PSO particle changes regardless of fitness value, whereas in DE the modification is only committed if a better or equal fitness value is obtained. The first approach allows an optimization algorithm to break free from local optima. DE uses the random selection of other vectors to create a similar effect. If we use the landscape analogy, as long as at least one DE vector is outside a depression in the landscape, but all the others are converging to the suboptimal minimum, DE has a probability to escape a local optima. Unlike PSO DE (in our configuration) does not use the global best in its calculations, sharing of information is completely distributed over the vectors.

Cost For each vector 3 other vectors are used, or restated we create 2 new vectors. Similar to PSO these calculations have a complexity linear in d, the dimensionality of the problem. The fitness function is called once per iteration per vector. Compared to PSO we therefore have the exact same evaluation complexity of O(nk).

Configuration CSRM uses a DE/rand/2/bin configuration. The DE/x/y/z notation reflects its main configuration, where x is the vector perturbed, y is the vectors used in the differential step and z is the crossover operation (binomial). This configuration is referenced [21] as one of the most competitive for multimodal problems with good convergence to the global optimum. Since we start from a probable local optimum the choice for a random vector instead of the global best vector also helps avoid premature convergence. This overview gives the values of each parameter used in CSRM's DE implementation.

- F = 0.6 : F should be in [0.4, 1] Large F values favor exploration, whereas small F values favor exploitation. The value of 0.6 is reported as good starting value. In our problem domain we already have a (sub) optimal solution which we wish to improve, so the risk of premature convergence is present, hence the small bias for exploration.
- Cr = 0.1 : The Cr values should be in [0,0.2] for separable functions, and [0.9, 1] for non separable functions. We cannot assume dependency between the constants, and therefore use a value of 0.1. This results in DE focussing alongside the axes of each dimension in its search trajectory.

Compared to PSO DE has a low parameter count, optimal values for these parameters can be found in literature [4]. The population size should be t * d with t in [2, 10]. Since we do not know d in advance, and to keep the comparison fair we set the population at 50, allowing for optimal values for up to 25 dimensions (constants). While DE has a small set of parameters, their effect is still quite pronounced. There exists implementations of DE that use self adapting parameters, but this is beyond our scope. It should be noted that CSRM's optimizer has a very small optimization budget (in evaluation cost) and each new problem has potentially new characteristics. We therefore chose for the most robust values and configuration.

5 Experiments

5.1 Benchmark problems

Recent work on the convergence of GP-based SR [15, 16] featured a set of benchmark problems that pose convergence problems for SR implementations. We reuse these problems in our work in order to study convergence of CSRM's implementation.

5.1.1 Problems

These problems use at most five features, when present to CSRM we do not give the algorithm this knowledge. In other words it assumes each problem is a function of 5 features which may or may not influence the expected outcome. This is an extra test in robustness for the algorithm, while also testing the algorithm's capability as a classifier.

$$1.57 + (24.3 * x_3)$$

$$0.23 + 14.2 * \frac{x_3 + x_1}{3.0 * x_4}$$

$$-5.41 + 4.9 * (\frac{x_3 - x_0 + \frac{x_1}{x_4}}{3 * x_4})$$

$$-2.3 + 0.13 * \sin(x_2)$$

$$3.0 + (2.13 * ln(x_4))$$

$$1.3 + 0.13 * \sqrt{x_0}$$

$$213.80940889 - 213.80940889 * e^{-0.54723748542 * x_0}$$

$$6.87 + 11 * \sqrt{7.23 * x_0 * x_3 * x_4}$$

$$\frac{\sqrt{x_0}}{\ln(x_1)} * \frac{e^{x_2}}{x_3^2}$$

$$0.81 + 24.3 * \frac{2.0 * x_1 + 3.0 * x_2^2}{4.0 * x_3^3 + 5.0 * x_4^4}$$

$$6.87 + 11 * \cos(7.23 * x_0^3)$$

$$2.0 - 2.1 * \cos(9.8 * x_0) * \sin(1.3 * x_4)$$

$$32 - 3.0 * \frac{\tan(x_0)}{\tan(x_1)} * \frac{tan(x_2)}{\tan(x_3)}$$

$$22 - 4.2 * ((\cos(x_0) - \tan(x_1)) * \frac{\tanh(x_2)}{\sin(x_3)}$$

$$12.0 - 6.0 * \frac{\tan(x_0)}{e^{x_1}} * (\ln(x_2) - \tan(x_3))$$

5.2 Operators

5.3 Constant Folding

5.3.1 Savings

5.3.2 Effect on convergence

5.4 Constant optimization

We look at the effect constant optimization using different algorithms has on different configurations of the tool. The measures used in the comparison are best fitness on training and test data, mean fitness on training and test data, and optimization cost.

5.4.1 Test problem

To verify our implementation for the optimizers we use a simple test problem and observe for each optimizer if it is able to optimize this instance to a known optimal value.

$$f(x_0, x_1, x_2) = 1 + x_1 * \sin(5 + x_2) * x_0 + (17 + \sin(233 + 9))$$

We give each optimizer a population of 50, 50 iterations and compare the results for 10 runs, displaying best value obtained, mean, and standard deviation of the fitness values compared to the known best value.

Best fitness In Figure 7 we see that DE outperforms PSO and ABC with several orders of magnitude. The best fitness value obtained was 2.22 e-16. As smaller but significant difference is present between PSO and ABC. This result is somewhat surprising given that fact that ABC is allowed to perform more evaluations in its configuration. From our previous discussion 4.2.2,4.2.3,4.2.1 we can conclude that for this test problem DE is clearly preferable as it obtains the best result at minimum cost. ABC has almost double the cost compared to PSO and DE, with PSO and DE having an equal cost in evaluations. The results on this testproblem do not necessarily mean that in the application of the three optimizers the results will be identical. Here we have a known optimal solution and want to observe how fast the optimizers converge to it. When we optimize evolved expressions we do not know what the optimal solution is. The problem statement is different, and so the convergence behavior is likely to differ as well. In Figures 8 and 9 we see that both the mean and standard deviation follow the same pattern as seen for the minimum fitness value with DE leading the others by several orders of magnitude. With all three distributions behaving similarly, this result provides a more solid foundation for our conclusions that for this problem DE is indeed the better optimizer.

5.4.2 Benchmark Problems

5.4.3 2 Phases

Figure 7: Logarithmic value of best fitness for each optimizer.

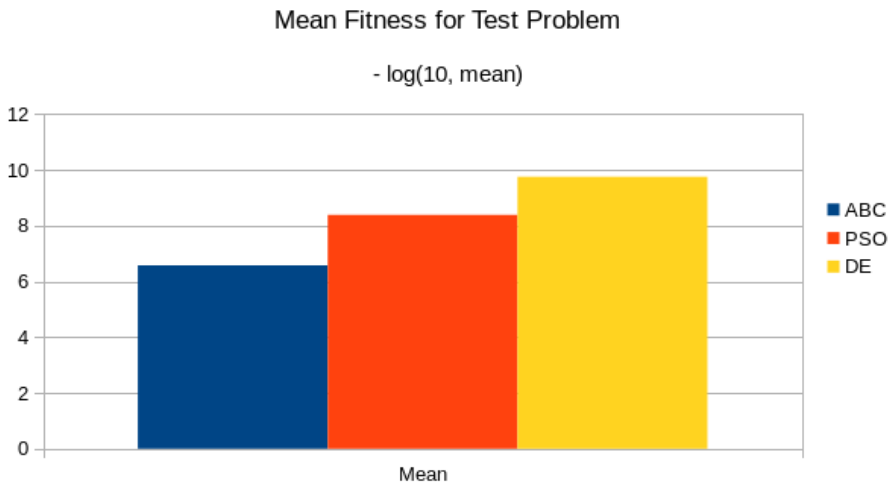


Figure 8: Logarithmic scaled mean fitness for each optimizer.

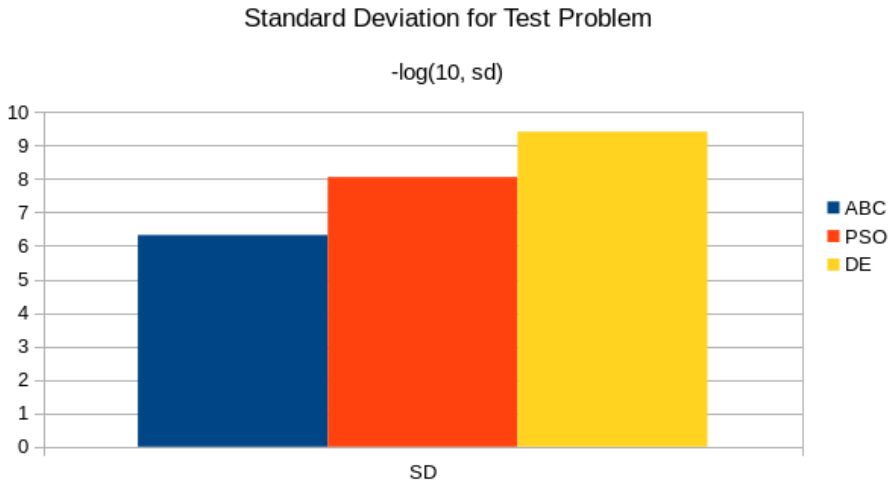


Figure 9: Logarithmic scaled standard deviation fitness for each optimizer.

Table 1: Results using the optimizer at the end of each phase after 2 phases.

Minimum fitness on training data

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
PSO	0.0	0.028090330794836027	0.014724476435230671	0.0008545866142538605	0.0	0	0.0	-0.0002525251249734239 4	0.005237137086905985	-0.0002363807373223250 32	0.060921433647570855	0.03417824633805988	0.00320814140466236	0.0	0.016638582591906 98
ABC	0	0.0	0.023251992312635417	0.006193090469415741	0.0	0	5.450069227952436 5e-05	3.722680786644172 5e-05	0.005190109277460775	-8.246521995014522e-05	0.09677501370548114	0.029553430956816618	0.01055038365227378 7	0.0	-0.000339184941374304 2
DE	0.0	0.021493115276747576	0.014295780309495 41	0.004307279089553 7505	0.0	0	-0.000156801157476293 97	0.000442742119690 8714	0.004471913346636 325	-0.000236349073350 0173	0.057339095932022 02	0.042559340376387 31	-0.00235427524789 78857	0.021493115276747 576	-0.000450079997018 07133

Minimum fitness on full data

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
PSO	0.0	0.08684762206260865	-0.0010297687963751745	-0.0008551385551067714	0.0	0	0.0	-0.0004673856202734 8607	0.006747476294210131	-9.605088174668985e-05	0.10344158698271044	0.016620982520120897	0.00226095710997653 58	0.0	0.027374351112392947
ABC	0.0	0.0	0.013029508684841096	0.004727172702580606 5	0.0	0	4.796539335161221e-05	3.830161529716491 5e-05	0.006914220665689252	3.826202192147576 4e-05	-0.08291826859342544	0.012840990613866898	-0.02710212803730 5717	0.0	-0.002651583281979 1196
DE	0.0	0.047933414716632 15	0.004495328823326 15	0.003183416376315 4733	2.220446049250313e-16	0	-0.000167321750825011 33	0.000470879832329679 3	0.005113604767726 98	-9.609635467844324e-05	-0.034100482513777 229	0.024601659376792 706	-0.00189626462543990 3	0.047933414716632 15	-0.001260920241817 00696

Gain in mean of 5 best expressions on training data

algorithm	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
PSO	0.0024317537312	0.214479082789	0.024415959453 3	0.0006446322162 75	0.0001361519751 84	7.099872105 58e-05	8.14636031861e-05	0.000189692831 44	0.002294555601 686	-0.00037432770112	0.06144289048 714	0.11929172701 6	0.00147351798865	0.181428435617	-0.009781022043 07
ABC	0.0023142462876 8	0.181428435617	0.04648104718 75	-0.00107484492601	8.56906438478e-05	1.97735707739e-05	0.000503667068672	0.000263800435 181	0.005703337993 17	-0.00023444823 1986	0.04937776014 76	0.07268815850 07	0.00491081059542	0.181428435617	-0.008977410828 17
DE	0.0022143353320 4	0.205846877316	0.028884678031 2	0.003156636955 1	0.000115675017 268	-4.58422394453e-05	-0.001220628287 7	-8.389835780 98e-06	0.005517386 1588	9.87758024436e-05	0.02210570 5529	0.07779425525 31	-0.00231712396414	0.205846877316	-0.008954231383 22

Gain in mean of 5 best expressions on full data

algorithm	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
PSO	0.0022007706161	0.31507068928	0.027516123542 3	0.000467696426923	0.000121901322701	5.18055280091e-05	0.000108492740646	-3.10009484427e-05	0.00441223591708	-0.000356614550643	0.028101739589	0.227717215934	0.000758522712758	0.224203645612	-0.006984279514 41
ABC	0.0020935371028 4	0.224203645612	0.053154516338 5	-0.000356178107312	7.44985729107e-05	3.93075261069e-06	0.000506949777787	0.000141153234319	0.009288303384841	-0.000210256554991	-0.07588409329 86	0.062181408029 8	-0.00327182850956	0.224203645612	-0.008034410865 94
DE	0.0020195981535 1	0.261125572494	0.031796495564 3	0.003498936049 78	0.000102330648102	-3.84522267255e-05	-0.001087151649 41	-4.595199451 75e-05	0.008519176946 69	2.77163295251e-05	-0.02664393418 89	0.211309995383	-0.00163824976308	0.261125572494	-0.014701817797 6

5.4.4 5 Phases

Fitness

Cost

5.4.5 10 Phases

Fitness

Cost

5.5 Conclusion

6 Related Work

7 Conclusion

8 Future work

References

[1] BANSAL, J. C., SINGH, P., SARASWAT, M., VERMA, A., JADON, S. S., AND ABRAHAM, A. Inertia weight strategies in particle swarm optimization. In *Nature and Biologically Inspired Computing (NaBIC), 2011 Third World Congress on* (2011), IEEE, pp. 633–640.

[2] BONYADI, M. R., AND MICHALEWICZ, Z. A locally convergent rotationally invariant particle swarm optimization algorithm. *Swarm Intelligence 8*, 3 (2014), 159–198.

[3] CLERC, M., AND KENNEDY, J. The particle swarm - explosion, stability, and convergence in a multidimensional complex space. *Trans. Evol. Comp 6*, 1 (Feb. 2002), 58–73.

[4] DAS, S., MULLICK, S. S., AND SUGANTHAN, P. Recent advances in differential evolution an updated survey. *Swarm and Evolutionary Computation 27* (2016), 1 – 30.

[5] EBERHART, R. C., AND SHI, Y. Comparing inertia weights and constriction factors in particle swarm optimization. In *Proc. of the 2000 Congress on Evolutionary Computation* (Piscataway, NJ, 2000), IEEE Service Center, pp. 84–88.

[6] EVETT, M., AND FERNANDEZ, T. Numeric mutation improves the discovery of numeric constants in genetic programming. In *in Genetic Programming. In* (1998), Morgan Kaufmann, pp. 66–71.

[7] GANSNER, E. R., AND NORTH, S. C. An open graph visualization system and its applications to software engineering. *SOFTWARE - PRACTICE AND EXPERIENCE 30*, 11 (2000), 1203–1233.

[8] GARDNER, M.-A., GAGN, C., AND PARIZEAU, M. Controlling code growth by dynamically shaping the genotype size distribution. *Genetic Programming and Evolvable Machines 16*, 4 (2015), 455–498.

[9] JENKS, G., ET AL. SortedContainers: Sorted list, dictionary and set types for Python, 2014–. [Online; accessed ¡today¿].

[10] KARABOGA, D., AKAY, B., AND OZTURK, C. *Artificial Bee Colony (ABC) Optimization Algorithm for Training Feed-Forward Neural Networks*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007, pp. 318–329.

[11] KARABOGA, D., OZTURK, C., KARABOGA, N., AND GORKEMLI, B. Artificial bee colony programming for symbolic regression. *Inf. Sci. 209* (Nov. 2012), 1–15.

[12] KENNEDY, J., AND EBERHART, R. Particle swarm optimization, 1995.

[13] KENNEDY, J., AND EBERHART, R. C. Particle swarm optimization. In *Proceedings of the 1995 IEEE International Conference on Neural Networks* (Perth, Australia, IEEE Service Center, Piscataway, NJ, 1995), vol. 4, pp. 1942–1948.

[14] KENNEDY, J. F., KENNEDY, J., EBERHART, R. C., AND SHI, Y. Swarm intelligence, 2001.

[15] KORNS, M. F. *Accuracy in Symbolic Regression*. Springer New York, New York, NY, 2011, pp. 129–151.

[16] KORNS, M. F. *A Baseline Symbolic Regression Algorithm*. Springer New York, New York, NY, 2013, pp. 117–137.

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

[18] LI, Z., WANG, W., YAN, Y., AND LI, Z. Psabc: A hybrid algorithm based on particle swarm and artificial bee colony for high-dimensional optimization problems. *Expert Systems with Applications 42*, 22 (2015), 8881 – 8895.

[19] O'NEILL, M., AND BRABAZON, A. Grammatical differential evolution. In *Proceedings of the 2006 International Conference on Artificial Intelligence, ICAI 2006* (Las Vegas, Nevada, USA, June 26-29 2006), H. R. Arabnia, Ed., vol. 1, CSREA Press, pp. 231–236.

[20] SPEARS, W. M., GREEN, D. T., AND SPEARS, D. F. Biases in particle swarm optimization. *Int. J. Swarm. Intell. Res. 1*, 2 (Apr. 2010), 34–57.

[21] STORN, R., AND PRICE, K. Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces. *Journal of Global Optimization 11*, 4 (1997), 341–359.