d(Tree)-by-dx: Automatic and Exact Differentiation of Genetic Programming Trees

Peter Rockett[1][0000−0002−4636−7727], Yuri Kaszubowski Lopes[1], Tiantian Dou[1], and Elizabeth A Hathway[2]

[1] Dept of Electronic and Electrical Engineering, University of Sheffield, Portobello Centre, Pitt Street, Sheffield S1 4ET, UK
[2] Dept of Civil and Structural Engineering, University of Sheffield, Mappin Street, Sheffield S1 3JD, UK
`p.rockett@sheffield.ac.uk`

Abstract. Genetic programming (GP) has developed to the point where it is a credible candidate for the 'black box' modeling of real systems. Wider application, however, could greatly benefit from its seamless embedding in conventional optimization schemes, which are most efficiently carried out using gradient-based methods. This paper describes the development of a method to automatically differentiate GP trees using a series of tree transformation rules; the resulting method can be applied an unlimited number of times to obtain higher derivatives of the function approximated by the original, trained GP tree. We demonstrate the utility of our method using a number of illustrative gradient-based optimizations that embed GP models.

Keywords: Genetic programming · Automatic differentiation · Optimization · Real-world applications.

1 Introduction

It is now widely accepted that genetic programming (GP) is capable of competitive application across a range of sciences and engineering [5]. One area that has not hitherto received much attention is the important topic of integration of GP into conventional optimization-based applications, such as control, that typically require the computation of derivatives for fast solution. Izzo et al [8] have recently listed a range of of diverse applications of GP that require derivatives for effective solution.

In particular, our specific interest is model predictive control (MPC) [3] where a dynamic predictive model of system behavior is used to optimize future inputs over a so-called *prediction horizon* extending many time steps into the future. MPC is especially suited to systems where there is a significant time lag between application of an input and an observable response, and has been widely used in the process industries although interest is growing for the indoor environmental control of buildings [14]. One of the major problems—and indeed costs—of MPC is the economic acquisition of a suitable predictive model, and GP, as a machine learning technique, is particularly attractive for this purpose. Hitherto, however, the ability to integrate GP predictive models into the fast, gradient-based optimizers conventionally used in MPC has been a significant barrier to the deployment of GP.

Potentially, so-called derivative-free conventional optimizers [4] can avoid the need for explicit derivatives although the solution times are invariably longer than for techniques explicitly based on derivatives; further, derivative-free solvers often internally approximate derivatives, a limitation we will expand upon later in this paper. In essence, the derivatives of the objective function to be optimized usually have to implicitly exist, at least up to second order.

Mousavi Astarabadi and Ebadzadeh [12] have calculated tree derivatives using finite difference approximations, but there is a well-known trade-off here between making the perturbation on the variable of differentiation too small and the accuracy being dominated by round-off errors, and making it too big and suffering large truncation errors. Unfortunately, the optimal value of perturbation typically varies across the domain.

Finally, stochastic search methods, such as differential evolution, particle swarm optimization, etc. strictly require no gradient information but tend to be too slow for real-time applications, such as control [6].

To expand the range of applications of GP, a means of exploiting fast, gradient-based optimizers is therefore highly desirable. This, in turn, requires a straightforward and reliable method of calculating the derivatives of GP trees, and this is the key contribution of the present paper.

Izzo et al. [8] have recently reported the application of truncated Taylor polynomials in the context of Cartesian genetic programming to calculate derivatives. These authors concede, however, that the "necessary algebraic manipulations" are "non trivial". The approach we present here, on the other hand, is much simpler. Indeed, we believe all of the concepts will be very familiar to the GP community.

In this paper, we describe a set of tree transformations that generate the partial derivative (with respect to a given variable) of a real function described by conventional tree-based GP. In Section 2, we describe the problem formulation, and we derive the necessary tree transformations in Section 3. We give practical implementation details in Section 4. Section 5 describes two examples of the application of our tree-differentiation technique for embedding GP within conventional, gradient-based optimizations. Possible future work, including extensions, is discussed in Section 6; Section 7 concludes the paper.

2 Problem Statement

A very general requirement for a real function—be it implemented by a GP or otherwise—to be differentiable is for it to be *analytic*. (More pedantically, an analytic function is infinitely differentiable [15].) Since GP tree mappings ($\mathbb{R}^N \to \mathbb{R}$) are simply compositions of the internal function nodes, and a composition of analytic functions is itself analytic [15], it follows that a GP tree will be analytic, and therefore (infinitely) differentiable, if each of the function nodes is analytic. At this stage, the principal challenge becomes apparent. The commonly-used set of function nodes in GP comprises: addition ('+'), subtraction ('−'), multiplication ('×') and some version of protected division (PD) defined by, for example:

$$\frac{f}{g}\begin{cases}\frac{f}{g} & g \neq 0\\ 1 & g = 0\end{cases} \tag{1}$$

where f, g are the values returned by the two subtrees of the PD operator.

Although '+', '−' and '×' are everywhere analytic, protected division is not. In particular, when $g = 0$ the limit defining the derivative does not exist. While GP systems using only a function set of $\{+, -, \times\}$ have been explored, the ability to divide some quantity into 'parts' appears to give greater expressiveness [10]. (More generally, Keijzer [10] criticizes the use of 'protected' operators in GP.) Fortunately, an elegant solution to the above problem has already been reported by Ni et al. [13] who replaced the protected division operator with an analytic quotient defined by

$$AQ(f, g) = \frac{f}{\sqrt{1 + g^2}}$$

that tends asymptotically to the quotient value $f/|g|$ when $|g| \gg 1$, but retains analyticity at $g = 0$. Although the principal motivation in [13] was to avoid problems with the PD operator returning huge values when $|g|$ was very small but strictly > 0, the differentiability of the operator was remarked upon in [13]. In consequence, in this work we replace the conventional protected division operator with the analytic quotient (AQ) operator.

3 Theory

Since the fundamental requirement for all function nodes to be analytic can now be met with a function set of $\{+, -, \times, AQ\}$ supplemented by a leaf node set of $\{x_i \in \mathbb{R}\ \forall i \in [1 \ldots n]$, and constants $\in \mathbb{R}\}$ where n is the dimensionality of the input vector $\mathbf{x}$, an arbitrary GP tree can be differentiated using repeated, recursive application of the standard chain rule of calculus for the differentiation of a composition of functions. Thus, where $F(x) = h(k(x))$:

$$F'(x) = h'(k(x)).k'(x) \tag{2}$$

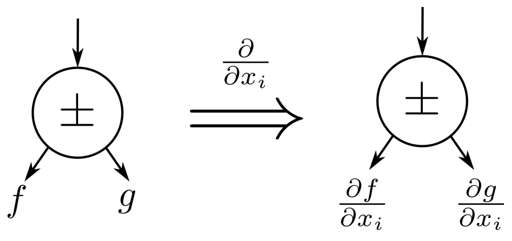
3.1 Internal Function Nodes

Differentiation of each of the internal function nodes follows straightforwardly from (2), and the chain rule and can be conveniently formulated as a transformation of the elements of the GP tree.

Addition/Subtraction The derivative can be written as:

$$\frac{\partial(f \pm g)}{\partial x_i} = f' \pm g' \tag{3}$$

which can be represented by the simple tree transformation in Fig. 1.



$\frac{\partial}{\partial x_i}$
$\pm$
f
g
$\pm$
$\frac{\partial f}{\partial x_i}$
$\frac{\partial g}{\partial x_i}$

Fig. 1. Addition/subtraction transformation

Multiplication Similarly, the derivative can be written as:

$$\frac{\partial(fg)}{\partial x_i} = f'g + fg' \tag{4}$$

which is represented by the tree transformation shown in Fig. 2.

Analytic Quotient Finally, the derivative of the analytic quotient can be written as (5), which can be rearranged as (6) in the form of other analytic quotient functions:

$$\frac{\partial\, AQ(f,g)}{\partial x_i} = f'(1+g^2)^{-1/2} - fgg'(1+g^2)^{-3/2} \tag{5}$$

$$= \frac{f'}{\sqrt{1+g^2}} - \frac{f}{\sqrt{1+g^2}}.\frac{g}{\sqrt{1+g^2}}.\frac{g'}{\sqrt{1+g^2}} \tag{6}$$

The practical choice between (5) and the less compact (6) is discussed further in Section 4, but for the present, the tree transformation corresponding to (6) is shown in Fig. 3.

3.2 Terminal Nodes

Any given GP tree can be differentiated by repeated, recursive application of the transformations shown above. Finally, however, the inputs to the transformed (differentiated) trees will reach the leaves of the tree. For completeness, the derivative of a variable is given by (7), and for a constant leaf node by (8).

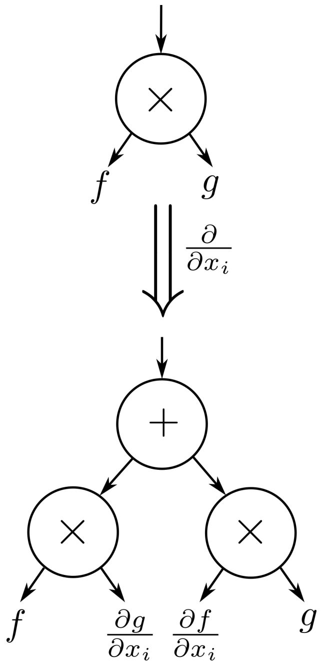

$\times$
f
g
$\frac{\partial}{\partial x_i}$
$+$
$\times$
$\times$
f
$\frac{\partial g}{\partial x_i}$
$\frac{\partial f}{\partial x_i}$
g

Fig. 2. Multiplication transformation

$$\frac{\partial x_i}{\partial x_j} = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases} \tag{7}$$

$$\frac{\partial c}{\partial x_i} = 0 \quad \forall i \in [1 \dots n] \tag{8}$$

4 Implementation

Since trees are differentiated by application of the chain rule of calculus, our implementation is correct by construction.

With reference to Section 3, it is clear that multiplication, and especially the AQ operator, result in significant growth in tree size compared to the original, undifferentiated tree. One AQ node in the original tree transforms to (at least) seven nodes in the differentiated tree. This tree growth is a direct consequence of the chain rule of calculus. (For reference, the derivative of a regular (unprotected) quotient operation would transform to five nodes.) The increased complexity of derivatives is well-recognized in numerical analysis, and many derivative-based solvers include either automatic differentiation libraries to generate derivatives from instrumented source code [1], or additional utilities to roughly check hand-generated derivatives using finite difference approximations.

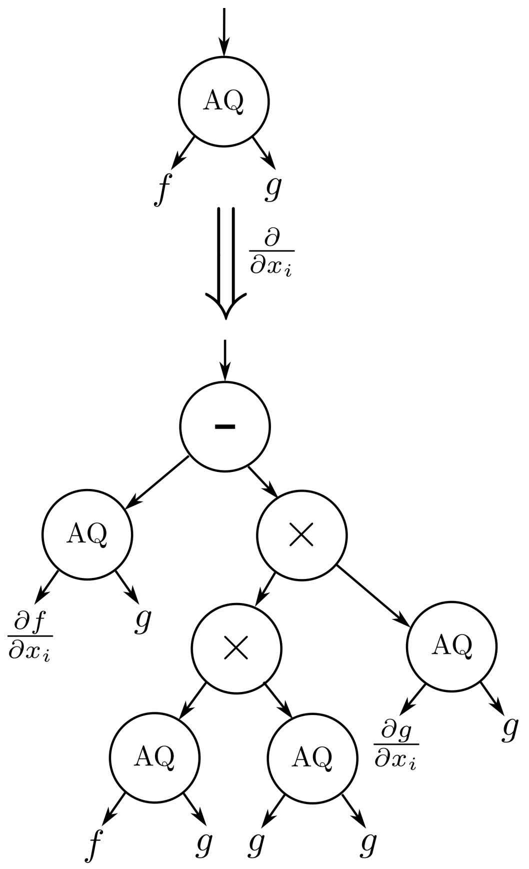

AQ
f
g
$\frac{\partial}{\partial x_i}$
–
AQ
×
$\frac{\partial f}{\partial x_i}$
g
×
AQ
AQ
AQ
$\frac{\partial g}{\partial x_i}$
g
f
g
g
g

Fig. 3. Analytic quotient transformation

In the context of the present work, the tree growth from differentiation poses a design choice: should we use the same function set and express the derivative of the AQ operator using the existing function set at the inevitable cost of some tree growth? Or should we augment the function set with a new type of 4-ary function node that embeds the derivative of AQ as a single node and takes f, g, $\partial f/\partial x_i$ and $\partial g/\partial x_i$ as inputs? This latter option would lead to less tree growth since the differentiated AQ node would be replaced by a single derivative node.

We have chosen the former route since expressing the derivative tree using only the original function set allows repeated application of the tree differentiation operation without limit to form higher derivatives; for example, the Hessian matrix, the elements of which comprise second-derivatives such as $\partial^2 f/\partial x_i \partial x_j$, is often invaluable in optimization problems. The alternative implementation of expressing an AQ derivative within the tree as a new 4-ary node would require the addition of further, special nodes to handle the derivatives of the derivative of an AQ operator. Generating third and higher order derivatives would cause further significant complications.

While we give an example later of the direct use of second-order derivatives in optimization, at the present time we can suggest no immediate application for third and higher-order derivatives; nonetheless, such quantities are frequently used in mathematical analysis [15]. The ability to generate higher derivatives, or to assure the analyticity of a GP model thus creates an enabling platform for possible future research.

Detailed implementation was in the form of a function that takes the GP tree as an argument together with the index in $\mathbf{x}$ of the variable of differentiation, and returns a tree generated by application of the tree transformations in Section 3. This newly created tree is completely independent of the original, undifferentiated tree and can be recursively evaluated in the normal manner to obtain the value of the derivative of the original tree at some arbitrary $\mathbf{x}$. Our initial implementation is in C++ and we reference both the original and differentiated trees using C(++) pointers to the trees.

5 Experiments

In order to fully demonstrate the efficacy of automatically-calculated tree gradients, we present two example use cases of embedding of GP within conventional optimization frameworks. Section 5.1 reports the use of first-derivative information only while Section 5.2 describes the inclusion of explicit Hessian information. Both cases would be typical of control or related applications of GP.

5.1 Gradient-based Minimization - I

As an initial demonstration, we have used the tutorial example in the documentation of the highly-regarded NLopt[3] nonlinear optimization library [9], and given in (9) and (10).

$$\underset{\mathbf{x}\in\mathbb{R}^2}{min}\sqrt{x_2} \tag{9}$$

$$\text{s.t.}\;\; 2x_1^3 - x_2 \leq 0 \;\;\text{and}\;\; (-x_1+1)^3 - x_2 \leq 0 \tag{10}$$

We selected ten evenly spaced points in both x_1 and x_2 to train three separate GP approximations to the objective function (9), and the two constraint functions (10). We have used a fairly standard GP framework with a population of 100 individuals, a hard depth limit of 8, and a fixed number of 10,000 function evaluations per function. (The exact details are unimportant—we have deliberately made no attempt to accurately learn the functions in (9,10) since our objective was not to try to reproduce the optimization results for this test problem,

[3] We have used NLopt version 2.5 downloadable from `https://github.com/stevengj/nlopt/archive/v2.5.0.tar.gz`

but rather demonstrate the utility of our tree differentiation approach.) Having (approximately) learned each of the three functions in (9,10), these trained trees were incorporated into the NLopt example program. We used the sequential least-squares quadratic programming (SLSQP) algorithm from NLopt [11] to solve the optimization problem, which requires derivatives of both the objective function and of the two constraint functions. Six separate trees implementing the derivatives in $x_{1,2}$ of each of the three functions described above were generated *at runtime*, and evaluated as demanded by the SLSQP algorithm to return the values of the derivatives. The optimization was set to terminate when the relative error of the optimized parameters fell below 10^{-4}, which required 27 function evaluations; the optimal solution was obtained at the point $(0.300489, 0.282026)$ and an objective function minimum of 0.256948.

(In contrast, the original problem in (9,10) had an optimum at $(0.3333, 0.2963)$, a function minimum of 0.5443, and required 53 iterations with the same termination criterion. The differences can, of course, be explained by the fact that we have solved an *approximation* of the original problem in (9,10) using GP trees.)

To judge whether our GP-generated solution was indeed a minimum, we generated 1 million random points in a 2D square of 0.1 on a side and centered on the solution point. We were unable to find any point in the neighborhood of the solution that had a lower objective function and that satisfied the constraints. We thus infer that the identified solution point is indeed a minimum (of that modified problem).

5.2 Gradient-based Minimization - II – Using Hessian Information

As a second example, we have used our GP differentiation framework within the Ipopt [16] large-scale interior-point algorithm for nonlinear programming that combines line-search and trust region methods [17]. We have used Ipopt version 3.12.12 freely downloadable from `http://www.coin-or.org/download/source/Ipopt`. We have used the example problem from Ipopt's documentation:

$$\min_{x_1,x_2,x_3,x_4 \in \mathbb{R}} x_1 x_4 (x_1 + x_2 + x_3) + x_3 \tag{11}$$

$$\text{s.t.} \quad x_1 x_2 x_3 x_4 \geq 25 \tag{12}$$

$$x_1^2 + x_2^2 + x_3^2 + x_4^2 = 40 \tag{13}$$

$$1 \leq x_{1,2,3,4} \leq 5 \tag{14}$$

which, in turn, is a standard, constrained nonlinear benchmark problem from the collection published by Hock and Schittkowski [7]. The problem has an optimal solution given by: $\mathbf{x}^* = (1.00000000, 4.74299963, 3.82114998, 1.37940829)$ with a minimum objective value of 17.014017140224134.

Ipopt is an interesting application of our method because, as well as needing gradient values, it requires the Hessian of the Lagrangian functional, $\nabla^2 f + \lambda_1 \nabla^2 g_1 + \lambda_2 \nabla^2 g_2$, where f is the objective (11), $g_{1,2}$ are the two constraint functions (13) and (14), and $\lambda_{1,2}$ are the Lagrange multipliers. If the Hessian information is not explicitly available, Ipopt approximates the Hessian of Lagrangian internally with some loss of accuracy. We are thus able to explore if our approach can produce comparable results to conventional coding of the objective and constraint functions, and their first and second derivatives.

Since both the objective and constraint functions in (11)-(14) are exactly representable with addition and multiplication operators, we have hand-constructed GP trees for the objective function (11) and the two constraint functions (13) and (14) only, and verified that these trees returned identical numerical results compared to direct C coding of the functions within floating-point round-off error. We have then embedded these hand-coded trees within the example Ipopt C code. Rather than hand-evaluate the necessary gradients and Hessians (as was done in the Ipopt example code), we have automatically generated these quantities by application of our tree differentiation procedures. Since the gradient vectors $\nabla f, g_{1,2}$ are each composed of four partial derivatives, each differentiation with respect to x_{1-4}, we generated—at *runtime*—a total of $4 \times 3 = 12$ trees implementing ∇f and $\nabla g_{1,2}$.

Similarly, we generated the 4×4 Hessian matrices $\nabla^2 f$ and $\nabla^2 g_{1,2}$ with a second round of applications of the procedure to ∇f and $\nabla g_{1,2}$, again at runtime. That is, we have differentiated the derivatives. Each element of each Hessian matrix required a separate GP tree making a total of $4 \times 4 \times 3$ trees to produce the second-order information. The generation of none of the first or second-order derivative information required any manual intervention (other than coding the invocations of the tree differentiations).

We then compared the GP-based optimization against the use of the hard-coded implementation of the objective, constraints, gradient, Lagrangian, and Hessian functions. Since manually calculating Hessian matrices is usually tedious and error-prone, Ipopt offers the facility to internally approximate this quantity albeit with some loss of accuracy. We thus also used this method to provide the second-order information within our GP-based implementation of the optimization to compare with exact, automatic generation of the Hessian by tree differentiation. We tried various combinations of calculating the necessary functions and their performances are shown in Tab. 1.

The basic comparison to be made is between the first and sixth rows of Tab. 1 since these show the results of implementing the objectives and constraint functions, their derivatives and Hessians using hand-coded C (id = 01), and implementing the objective and constraint function as GP trees and then automatically generating the first and second-order derivative information at runtime by tree differentiation (id = 06). The differences between the two objective values obtained (using standard IEEE 754:1985 double-precision floating point arithmetic with roughly 16-17 decimal digits) occurs in the least-significant digit and is almost certainly due to rounding error. To summarize this sub-section, we have demonstrated that our automatic tree differentiation procedure is able to produce accuracies that differ from the results of hand-coded evaluation by what we believe to be rounding error alone.

Table 1. Results for the gradient-based minimization - II. Comparison between the use of hand-coded (CD) functions, the use of GP trees, and the use Ipopt's Hessian approximation (HAP). The table shows the method of calculating: the objective function (f), constraints (g), the gradient of the objective function (∇f), the Jacobian of the constraints ($\boldsymbol{J}$), and the Hessian ($\boldsymbol{H}$). The final column shows the difference in the objective value compared to the fully hand-coded solution.

id	f	g	∇f	$\boldsymbol{J}$	$\boldsymbol{H}$	Objective	Difference
01	CD	CD	CD	CD	CD	17.014017140224134	n/a
02	GP	CD	CD	CD	CD	17.014017140224134	0
03	GP	GP	CD	CD	CD	17.014017140224137	$+3 \times 10^{-15}$
04	GP	GP	GP	CD	CD	17.014017140224134	0
05	GP	GP	GP	GP	CD	17.014017140224137	$+3 \times 10^{-15}$
06	GP	GP	GP	GP	GP	17.014017140224137	$+3 \times 10^{-15}$
07	CD	CD	CD	CD	HAP	17.014017140224176	$+4.2 \times 10^{-14}$
08	GP	CD	CD	CD	HAP	17.014017140224176	$+4.2 \times 10^{-14}$
09	GP	GP	CD	CD	HAP	17.014017140224176	$+4.2 \times 10^{-14}$
10	GP	GP	GP	CD	HAP	17.014017140224180	$+4.6 \times 10^{-14}$
11	GP	GP	GP	GP	HAP	17.014017140224176	$+4.2 \times 10^{-14}$

The first six rows of Tab. 1 show various combinations of using hand-coded functions, derivatives and Hessians and the corresponding quantities evaluated using GP trees. Any differences that do exist also appear due to rounding errors rather than issues with the accuracy of the GP derivative trees.

Rows id = 07 to id = 11 show the results of using Ipopt's internal approximation of the Hessian information. The differences compared to row id = 01 are all very similar and would thus all appear dominated by the errors due to Ipopt's internal Hessian approximation. A noteworthy point here is that hand-coding Hessian information is well-known to be a tedious and error-prone process due to the increasing complexity of the derivative-of-derivative expressions. Using automatic tree differentiation, however, there is a negligible cost to exact evaluation of the second-derivative information.

To summarize this sub-section, we have demonstrated that our automatic tree differentiation procedure is able to produce accuracies that differ from the results of hand-coded evaluation by what we believe to be rounding error alone.

6 Discussion and Future Work

Although the analytic quotient operator (AQ) [13] allows us to produce the transformation in Section 3.1, it is almost certainly not the only suitable function. Other analytic, quotient-type operators may be preferable, but the property of AQ of being able to construct derivatives of any order without extending the function set is very attractive for easily generating higher-order derivatives. Consideration of other quotient options is an area for future work.

Another area for future work is extension to additional function nodes (e.g. circular and other transcendental functions). Extension to sine and cosine would appear straightforward since these generate complementary derivatives. That is, the derivative of a sine function is a cosine, and vice versa. Similarly, the derivative of an exponential function is another exponential. The consequence is that forming higher derivatives by repeated transformation of already-calculated derivative trees is straightforward, and would not require extension of the GP function set. Other functions commonly used in GP might be more problematic.

We have framed the present paper in terms of tree transformations, but automatic differentiation (AD) of computer code has received considerable attention—see [1], for example, for a review. AD is, of course, also based on the chain rule of calculus but allows users to generate derivatives of expressions in conventional computer code by instrumenting that code, and typically using a pre-processor system to substitute compilable code implementing the required derivatives. The discriminating factor between AD and the present work is that GP trees are not usually manually programmed in conventional languages, but rather exist as trained tree models in the computer's memory. Our automatic differentiation operates directly on the in-memory data structures. Nonetheless, there may be opportunities for cross fertilization with the AD literature. In this context, Baydin et al. [2] have recently reviewed the links between AD and machine learning.

One major area of our future work will be to exploit GP tree derivatives in model predictive control (MPC) [3], as set out in Section 1. Results will be published elsewhere.

7 Conclusions

In this paper, we have introduced a series of tree transformations that can be applied recursively to generate independent trees for the evaluation of the derivative of the function implemented by the original tree. Since the differentiated tree can be expressed in terms of the original function set, we can apply the tree differentiation procedure any numbers of times to produce, again independent, GP trees, that implement higher-order derivatives.

We have demonstrated the application of our tree differentiation procedure on two representative constrained, non-linear optimization problems. These demonstrators involved conventional optimization in which both the objective and constraint functions were implemented with genetic programming trees.

The present paper thus makes an important contribution to extending the application of genetic programming to novel, real-world problems, especially control.

Acknowledgements

We gratefully acknowledge supported by the UK Engineering and Physical Sciences Research Council (EPSRC) under grant EP/N022351/1.

References

1. Bartholomew-Biggs, M.C., Brown, S., Christianson, B., Dixon, L.C.W.: Automatic differentiation of algorithms. Journal of Computational and Applied Mathematics **124**, 171–190 (2000). https://doi.org/10.1016/S0377-0427(00)00422-2
2. Baydin, A.G., Pearlmutter, B.A., Radul, A.A., Siskind, J.M.: Automatic differentiation in machine learning: A survey. Journal of Machine Learning Research **18**(1), 5595–5637 (2017), `http://jmlr.org/papers/v18/17-468.html`
3. Camacho, E.F., Bordons, C.: Model Predictive Control. Springer, London, 2^{nd} edn. (2004)
4. Conn, A.R., Scheinberg, K., Vicente, L.N.: Introduction to Derivative-Free Optimization. Society for Industrial and Applied Mathematics (2009). https://doi.org/10.1137/1.9780898718768
5. Gandomi, A.H., Alavi, A.H., Ryan, C. (eds.): Handbook of Genetic Programming Applications. Springer Nature, Cham (2015). https://doi.org/10.1007/978-3-319-20883-1
6. Goldberg, D.E.: Genetic Algorithms in Search, Optimization and Machine Learning. Addison Wesley, Reading, MA (1989)
7. Hock, W., Schittkowski, K.: Test Examples for Nonlinear Programming Codes, Lecture Notes in Economics and Mathematical Systems, vol. 187. Springer-Verlag, Berlin/Heidelberg (1981). https://doi.org/10.1007/978-3-642-48320-2
8. Izzo, D., Biscani, F., Mereta, A.: Differentiable genetic programming. In: 20^{th} European Conference (EuroGP 2017). pp. 35–51. Amsterdam, The Netherlands (2017). https://doi.org/10.1007/978-3-319-55696-3_3
9. Johnson, S.G.: The NLopt nonlinear-optimization package. http://ab-initio.mit.edu/nlopt (2019)
10. Keijzer, M.: Improving symbolic regression with interval arithmetic and linear scaling. In: Ryan, C., Soule, T., Keijzer, M., Tsang, E., Poli, R., Costa, E. (eds.) European Conference on Genetic Programming (EuroGP 2003). pp. 70–82. Essex, UK (2003). https://doi.org/10.1007/3-540-36599-0_7
11. Kraft, D.: Algorithm 733: TOMP–Fortran modules for optimal control calculations. ACM Transactions on Mathematical Software **20**(3), 262–281 (1994). https://doi.org/10.1145/192115.192124
12. Mousavi Astarabadi, S.S., Ebadzadeh, M.M.: Avoiding overfitting in symbolic regression using the first order derivative of GP trees. In: Genetic and Evolutionary Computation Conference (GECCO Companion 2015). pp. 1441–1442. Madrid, Spain (11-15 July 2015). https://doi.org/10.1145/2739482.2764662
13. Ni, J., Drieberg, R.H., Rockett, P.I.: The use of an analytic quotient operator in genetic programming. IEEE Transactions on Evolutionary Computation **17**(1), 146–152 (February 2013). https://doi.org/10.1109/TEVC.2012.2195319
14. Rockett, P., Hathway, E.A.: Model-predictive control for non-domestic buildings: Critical review and prospects. Building Research & Information **45**(5), 556–571 (2017). https://doi.org/10.1080/09613218.2016.1139885
15. Rudin, W.: Principles of Mathematical Analysis. McGraw-Hill (1976)
16. Wächter, A., Biegler, L.T.: On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. Mathematical Programming **106**(1), 25–57 (2006). https://doi.org/10.1007/s10107-004-0559-y, `http://www.optimization-online.org/DB_FILE/2004/03/836.pdf`
17. Waltz, R.A., Morales, J.L., Nocedal, J., Orban, D.: An interior algorithm for nonlinear optimization that combines line search and trust region steps. Mathematical Programming **107**(3), 391–408 (2006). https://doi.org/10.1007/s10107-004-0560-5