Dissipative Arithmetic

William B. Langdon
Department of Computer Science
University College London
Gower Street, London WC1E 6BT, UK
www.cs.ucl.ac.uk/staff/W.Langdon

Large arithmetic expressions are dissipative: they lose information and are robust to perturbations. Lack of conservation gives resilience to fluctuations. The limited precision of floating point and the mixture of linear and nonlinear operations make such functions anti-fragile and give a largely stable locally flat plateau a rich fitness landscape. This slows long-term evolution of complex programs, suggesting a need for depth-aware crossover and mutation operators in tree-based genetic programming. It also suggests that deeply nested computer program source code is error tolerant because disruptions tend to fail to propagate, and therefore the optimal placement of test oracles is as close to software defects as practical.

Keywords: information loss; irreversible computing; entropy; evolvability; arithmetic; software mutational robustness; optimal test oracle placement; evolution of complexity; data dependent computational irreducibility; effective computational equivalence; experimental mathematics; algorithmic information dynamics

1. Introduction

Genetic programming (GP) [1] aims to evolve programs from scratch. Typically a program is represented as a tree structure, to which various operations are applied, such as subtree replacement and subtree exchange with another program. In the following experiments, we observe that for arithmetic expressions, many such changes do not change the program's output and find this is often due to the imprecision of floating-point arithmetic. This means that the runtime disruption caused by such a change did not propagate to the root of the GP tree that represents a given program, thus hiding a possible error. By sampling uniformly the space of large arithmetic expressions composed of the four common operators (+, −, × and ÷), we show changes usually do not impact expressions' values, and so large arithmetic expressions are resilient to change. Furthermore, even testing as many as a thousand test points (uniformly selected in the range −1.0 to +1.0), the disruption caused by changes to the rational functions on average penetrates only about 60 levels of nesting in a GP tree, and so changes are often invisible outside the expression. (The median impact of disruption is about 16 genetic programming tree levels, rather than 60, if the expression is not protected against divide by zero, see Section 6.) With fewer tests (i.e., a weaker test oracle), changes impact fewer levels. However, with such a large number of tests, chance disruption close to the outermost part of the expression (the root node) may indeed have a sizeable effect. The effects of the introduced error progressively fail to propagate across the expression, as the four computer arithmetic operators are dissipative.

In an idealized infinite-precision computer, in some cases such small changes might be visible. However, if the injected perturbation is sufficiently far from the top of the expression, real computer effects, such as limited precision and rounding error, tend to act to smooth away such changes.

From an evolutionary point of view [1, 2], large complex expressions are robust and present a smooth fitness landscape where many mutations have no discernible impact. While we deal exclusively with arithmetic expressions, there is growing evidence that this is true of programming in general [3, 4].

In the next two sections, we continue the introduction. First, to consider the implications for software engineering (Section 2.1) and very long-term digital evolution of complexity (2.2). Then Section 3 describes the connected areas of computational irreducibility (3.1), experimental mathematics (3.2) and algorithmic information dynamics (3.3). Sections 4 and 5 detail how we sample the space of large nested expressions and their neighbors. In the two experimental sections (Sections 6 and 7), numbers and operations are represented as floating-point numbers and operations. We create large arithmetic expressions, make small changes to them and trace the impact of the change. In most cases, the impact dies away before it can affect anything outside the expression.

2. Motivation

Interest in large expressions stems from two concerns. First, the test oracle placement problem [5], and second, the evolution of complex behavior in long-term evolution experiments (LTEE) [6].

2.1 Information Loss in Deeply Nested Software Impacts Testing

In software engineering, the test oracle problem [7] is well known. However, there has been less research on where to place test oracles. Essentially the oracle problem is: when testing software, not only must we have test inputs for the code being tested but we must have an oracle that “knows” the correct answer. For example, we might have test strings “A” and “B” for a string concatenation program and a test oracle that checks the output “AB” is generated. The oracle itself can sometimes be somewhat automated, for example, using metamorphic testing [8] and implicit oracles [7]. For example, a program to find the cube root might check that the return value when multiplied by itself three times is (within suitable precision bounds) the same as the test input [9]. Implicit oracles include: Did the program crash? Did the program complete in a reasonable time? Did it return a value at all?

In terms of these experiments, we find that the more deeply nested an error is, the more chance that it will not be visible at the program’s output. This suggests that the best place to check to see if the software being tested contains an error is as close to the error as possible. In terms of practical software, rather than the pure functions used here, we see in many circumstances software engineers already try to do this with unit testing. In unit testing, values calculated in small program modules are checked immediately, rather than after the whole program has finished. These results motivate this and further recommend, where feasible, inserting test oracles inside software units. With increasing use of automatic testing, it may be feasible to use many closely spaced test oracles, which thus have a higher chance of detecting errors.

2.2 Excessive Robustness of Deep Trees Hampers LTEE Evolution of Complexity

Recently we have shown that over many thousands, even a million generations, artificial evolution, specifically GP [1, 2], can continue to find improvements, but that the rate of improvement may slow [10, 11]. As expected, the size of the programs also grows. This led to this investigation to see if the reduction in the speed of evolution can be tied to the nature of the evolving artifacts. Essentially, we find that the GP system evolves into a very robust region of the search space, where most small moves in the space have absolutely no impact on performance. We suggest that this relates to the depth of the programs rather than their size. Should this occur in a GP system, most children produced by conventional GP genetic operations would have the same fitness as their parents and the population would converge, in the sense that even though each tree is unique, everyone has the same fitness and there is no useful evolution until a random lucky genetic change near the output of a tree disrupts fitness [12, 13].

3. Connected Areas

3.1 Data-Dependent Computational Irreducibility

We can view arithmetic expressions as being computations. Indeed, the trees shown in Figures 1, 3, 5, 8 and 12 are programs or subprograms that recursively evaluate the value of an expression. Usually they can be reduced to simpler expressions; that is, they are not computationally irreducible [14, 15]. For example, the lower part of Figure 1 (SUB (SUB 0.211 (ADD X -0.01)) X) can be reduced to 0.221-2X. Some expressions or subexpressions are always reducible. For example, $0.211 - -0.01$ should always be 0.221, and similarly $-X+-X$ should always be $-2X$. However, sometimes computations can be reduced using knowledge of their input data. For example,

$$x \times (((0.897 + -0.875) \times ((0.211 - x + -0.01) - x)) \div -0.132) - 0.995$$

(see again Figure 1) can be reduced to 0 if it is known $x = 0$. That is, programs might be computationally irreducible in general but computationally reducible for some datum or set of test input data.

In Sections 6 and 7, we use test cases to show in many circumstances large expressions are effectively computationally equivalent (i.e., on the test cases) to other large expressions that are slightly different from them. That is, the larger expression is not computationally irreducible for at least some test data, as it gives the same results as the smaller expression.

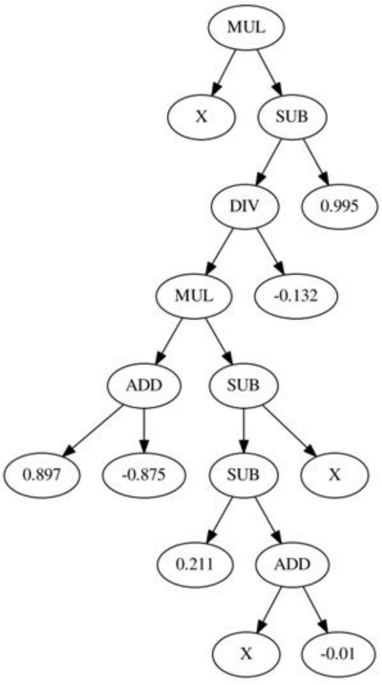


Figure 1. Example subexpression
$x \times (((0.897 + -0.875) \times ((0.211 - x + -0.01) - x)) \div -0.132) - 0.995$
as a tree (fun 5 from Table 1). Value of expression is given by the root node, here MUL, at the top. Plotted as yellow line in Figure 2.

Test	Depth		Change		
	Max	Mean	depth	size	function
0	435	245.7	254	1	X
replaced by fun 0				1	X
1	386	202.7	377	1	X
replaced by fun 1				1	X
2	235	125.6	141	3	(SUB X -0.263)
replaced by fun 2				1	−0.615
3	365	186.2	330	1	−0.67
replaced by fun 3				1	X
4	305	140.9	102	1	X
replaced by fun 4				1	−0.052
5	337	197.9	314	1	X
replaced by fun 5				17	(MUL X (SUB (DIX (MUL (ADD 0.897 -0.875) (SUB (SUB 0.211 (ADD X -0.01)) X)) -0.132) 0.995))
6	398	193.5	216	1	0.015
replaced by fun 6				15	(MUL (DIX (MUL (MUL X -0.803) (SUB (MUL 0.255 (DIX X X)) 0.389)) -0.756) -0.546)
7	358	156.8	152	1	−0.725
replaced by fun 7				13	(SUB (DIX X (DIX (SUB -0.016 (DIX -0.003 (DIX -0.399 -0.194))) 0.436)) 0.24)
8	283	145.6	76	1	X
replaced by fun 8				1	0.12
9	444	233.0	46	9	(SUB X (ADD (ADD -0.272 X) (SUB X X)))
replaced by fun 9				1	X

Table 1. Ten changes. Left: depth and average depth of each of the tree's nodes for 10 sampled trees, each representing arithmetic expressions with 25 001 nodes (Section 4). Right: uniformly selected changes (Section 5). Column 4 shows the depth of the uniformly chosen change site, while columns 5 and 6 show the removed subexpression and its replacement. DIX is either normal or protected division, depending on the experiment. (See also Figures 1 and 2.)

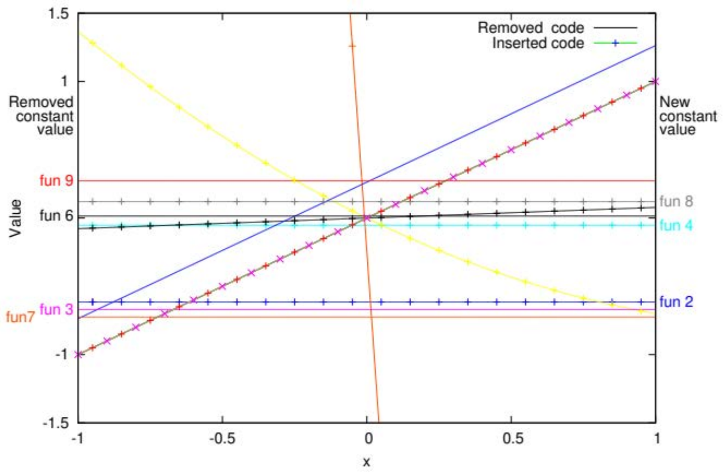


Figure 2. Eight pairs of changes plotted as functions of x. By chance, changes to functions 0 and 1 both replace x with x and are not plotted (see Table 1). Inserted subexpressions are plotted with lines and crosses. Horizontal lines indicate constants. Labels on the left margin indicate constant values that are removed. Labels on the right indicate constants that are inserted. In three examples, x is removed and in two, x is inserted. These are plotted on top of each other along the diagonal. In fun 7, the constant -0.725 is replaced by a randomly chosen linear function $-29.9835x - 0.24$ (near vertical line).

3.2 Experimental Mathematics

Experimental mathematics [14] is a branch of mathematics that instead of using only pure reasoning uses experimental investigation to gain insight, which may then lead to formal proof. For example, Simon Plouffe's 1995 discovery of the BBP formula [16] for the binary digits of π was achieved by computerized search followed by formal proof. In the mathematical tradition, we need to proceed with caution, as there are situations where formulas are initially correct in the numerical sense for the first 10 000 cases, but then found to fail for the 14 235th case [17]. In our experiments we are on firm ground since we are specifically targeting 32-bit precision. Although it would be nice to extend these results with formal proofs, the mathematics of floating-point arithmetic, overflow and rounding errors is notoriously difficult.

3.3 Algorithmic Information Dynamics

Algorithmic information dynamics [18–20], based on algorithmic "Solomonoff" probability, generates *programs* and so is more general than functions. In Section 4 onward, we deal with perturbations to functions (which are a proper subset of the set of *algorithmics*).

Notice algorithmic probability leads directly to a strong preference for generating simpler models and so follows Occam's razor, in contrast to GP, where regularization often needs to be added as part of bloat control [2].

4. Uniformly Sampling Large Arithmetic Expressions

All the experiments sample uniformly the space of expressions with 12 500 arithmetic operators [21]. As all four operators have two inputs, the expressions are binary trees with 12 500 internal nodes and 12 501 leaves (or external nodes). For a particular input x, the value of the whole expression is the value calculated by the operator that is the root node of the tree. (Figure 3 contains an example expression.)

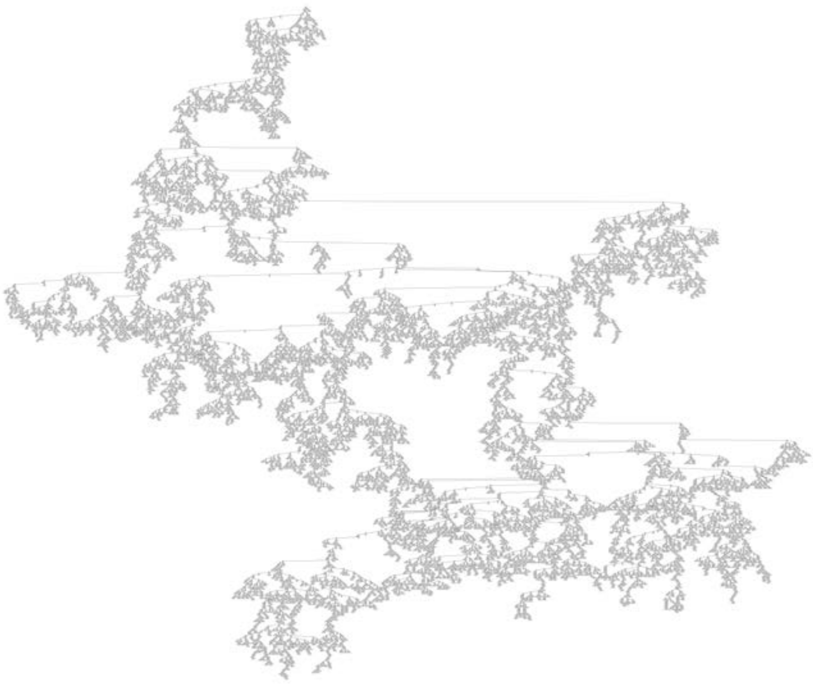


Figure 3. Expression 0 presented as a binary tree of 25 001 nodes (depth 435). Root node at top.

We start with a bare tree. It has been chosen uniformly at random from all the binary trees of size exactly 25 001. We chose the total tree size to be 25 001 (internal + external nodes), as such trees on average have depth close to $\sqrt{2\pi|\text{size}|} \approx 400$ (Flajolet and Oldyzko [22]), and previous work with evolved trees showed that, on average, typically the impact of mutations was lost after traversing about 100 functions [23].

We then convert it into a program by labeling each internal node with a function name (chosen at random) and label each external node with the name of a leaf. The leaves are either the input variable x or one of the constants. We use GPquick [24]. GPquick limits the number of constants to 250. In this experiment, the constants all have three digits, for example, -0.123, and are sampled from the range $[-1, 1]$. Before the experiment starts, we select (without replacement) 250 such constants, which we use in the (program) tree.

As a reminder, each binary tree's size is 25 001, and so it contains 12 500 functions and 12 501 leaves. For the leaves we only have x and 250 as constants, so we are bound to reuse. That is, the tree will have multiple copies of some constants and multiple copies of x. We choose to use x half the time, so on average, each tree will contain about 6250 copies of x in its leaf nodes. These will be scattered randomly.

We repeat the procedure 10 times and report results for 10 such randomly sampled trees. By chance, none of the special values -1.0, 0, or $+1.0$ are included. However, as we shall see in Section 6, the special value 0 is often present within the sampled arithmetic expressions (i.e., subtrees), due to subexpressions like $x - x$.

5. Sampling Changes

A site for each change is selected uniformly from each large expression. The subexpression at that location is removed and replaced by another subexpression. The inserted subexpression is similarly chosen uniformly from a large expression of the same size (i.e., 25 001 nodes). Table 1 describes the selected changes. The changes are plotted in Figure 2. The same changes are used in both the normal (Section 6) and the protected division (Section 7) experiments. Notice, by chance, change 9 is closest to the root node of its expression.

Each change is subsequently evaluated on a set of 1001 test cases, that is, values for x. These are selected uniformly in the range $[-1, 1]$ every 0.002 (e.g., 0.124). The value obtained at the root node of the unmodified tree is taken as ground truth.

6. Experiment I: Impact of Divide by Zero

The 10 created expressions (without mutation) contain on average 3125 divisions (i.e., $12\,500/4$) and about 195.3 $x - x$ (i.e., about

12 500 / (4×4×4)). Thus we encounter divide by zero. In floating-point arithmetic, this is commonly (as here) handled by division returning a special value representing ± infinity. All four floating-point operators deal with inf appropriately. That is, typically, if one of their arguments is infinity, then so too is their output. Thus infinity is propagated up the tree and typically the (floating-point) expression as a whole yields infinity. The output of the 10 selected large expressions is plotted for the input *x* range −1.0 to +1.0 in Figure 4. Eight of the 10 selected large expressions are shown in Figure 5. None of the 10 always give finite numbers, although one gives values between −100.0 and +100.0 on many test cases and one gives zero in all but one test case (where it gives -nan). In Section 7 we deal with this by "protecting" division from divide by zero errors.

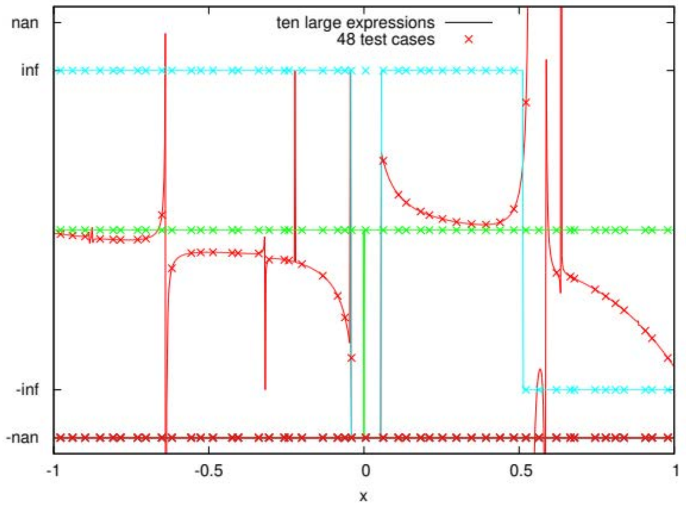
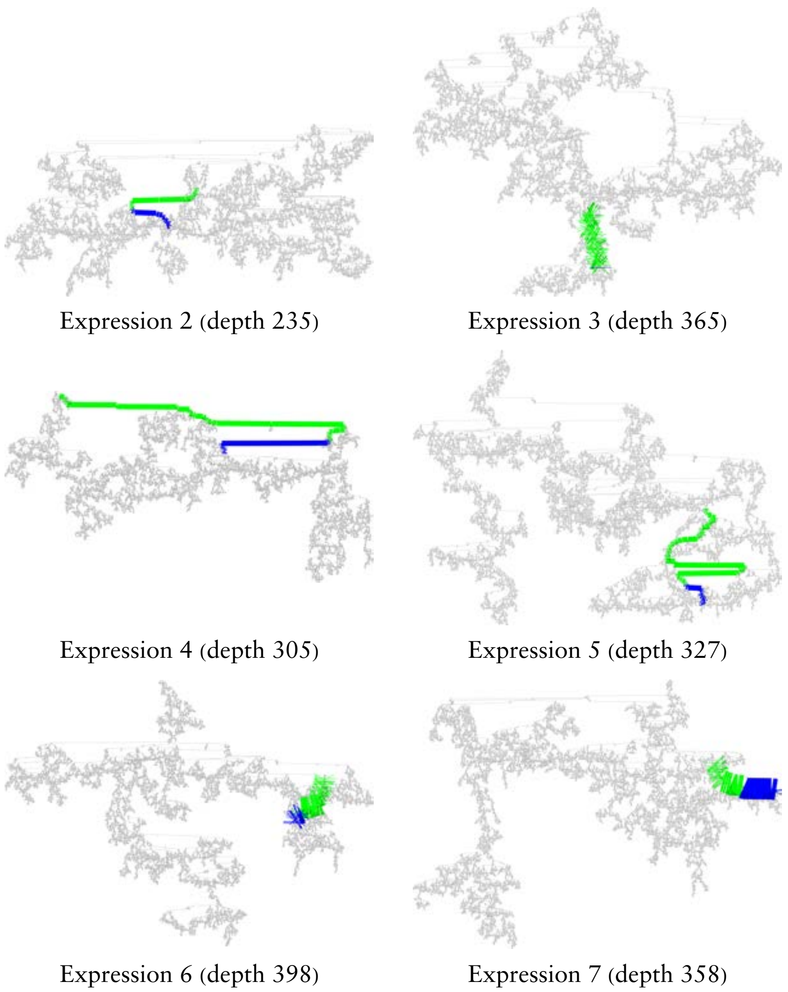


Figure 4. Ten large uniformly selected floating-point functions. Vertical axis has been linearly rescaled to plot very different output ranges on the same axis. Seven give -nan. Tree 4 gives either ±inf or -nan (light blue). Tree 1 gives either ±0 or -nan (green) and Tree 0 gives mostly numbers between −100 and +100, although it too can give ±inf and -nan (red).

Another common situation is multiply by zero. In floating-point arithmetic, multiply by zero yields ± zero. (For most purposes, the two types of floating-point zero are equivalent.) So again, multiply by zero destroys all the information about its other argument. However, zero is not as "sticky" as infinity, and the linear operators (+ and −) can readily transform zero into other numbers. Hence, typically, large trees do not yield zero for all inputs. With multiply by zero, it is easy to see why disruption to a subexpression can become invisible externally.

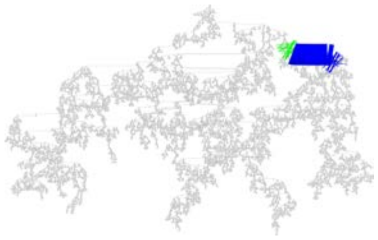

Expression 2 (depth 235) Expression 3 (depth 365)

Expression 4 (depth 305) Expression 5 (depth 327)

Expression 6 (depth 398) Expression 7 (depth 358)

Figure 5. (*continues*)

Each expression underwent a random mutation (i.e., disruption), as described in the previous section. None of the disruptions changed the value of the 10 expressions at the root node, yet within the tree changes are observed to the various subexpressions where the change took place. Figure 2 shows uniformly selected small changes to the 10 large functions. We need only plot the changes once, even though we use protected division in Section 7, as the small functions are not only syntactically the same in both this section and in Section 7, but (in these data ranges) they are also semantically the same. That is, Figure 2 applies to this section and to Section 7.


Expression 8 (depth 283)

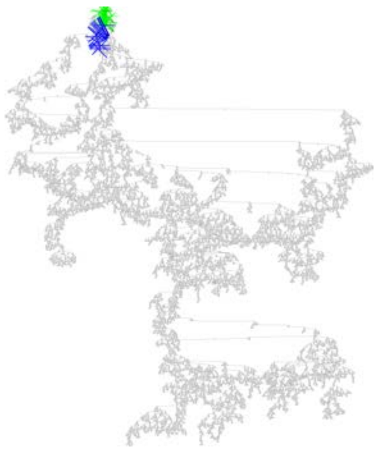

Expression 9 (depth 444)

Figure 5. Expressions 2 through 9 presented as binary trees of 25 001 nodes. Root nodes at top. Color indicates disrupted nodes. Blue nodes show subexpressions where at least one test case produces a different internal value as a result of the change with both normal and protected division. Green nodes (uppermost shaded nodes) show subexpressions where at least one test case is different when using division protected against divide by zero. Note here ordinary division is disrupted less than protected division.

6.1 Examples of Error Hiding: Multiply by Zero and Infinity and in General

Here we discuss how an error might not propagate to the root node, then present in detail how disruption failed to propagate in our 10 example trees.

If a leaf 0.125 is changed to 0.525, then it is as if an error of 0.4 is injected into the expression at that point. In a small expression, this might be easily observed. In a large expression, the 0.4 error is transformed by each function it passes through. In general, the error may become bigger or smaller. However, if it encounters a multiply by zero, the × result will be 0 regardless of the error. At this point, the error has vanished.

Notice the same can happen if the error encounters an infinity. For example, $a + \mathsf{inf} = \mathsf{inf}$ and also $(a + \text{error}) + \mathsf{inf} = \mathsf{inf}$. So again, the error vanishes and cannot be observed externally.

As we will see, multiply by zero and arithmetic on infinity are not the only reasons why a change may not affect the total result of an expression. Even under ideal conditions, floating-point arithmetic loses about half a bit of precision at each operation. So disruption is progressively suppressed. As the expressions are hierarchical, once disruption on a test case is lost (i.e., an internal function yields the same answer before and after the change), it cannot be reintroduced higher in the tree (i.e., the number of disrupted test cases falls monotonically). This continued interference of finite computing means the impact of even large errors can be totally lost in large expressions. Next, we discuss disruption propagation failures in our examples.

Figure 6 shows the fall in disruption of test cases as we move away from the disruption (i.e., as we move up the tree toward the root node). Figure 7 shows the same thing. But instead of counting the number of test cases that are not identical, Figure 7 plots the average difference before and after the change in the values inside each of the large expressions.

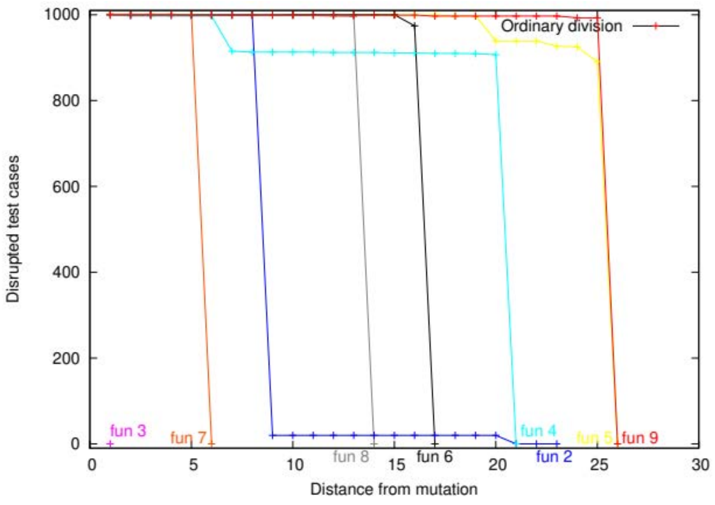


Figure 6. Fall in impact of eight changes with distance from disruption (functions 0 and 1 cause no disruption as both replace x with x and so are not plotted). Colors are the same as in Figure 2 and others.

For two changes (fun 0 and fun 1), by chance, there is no disruption, as they both replace x by itself (not plotted). For five changes, disruption is halted by encountering an infinity and in two more by a multiply by zero (see string of blue nodes in Figure 5). Even in the remaining example, fun 7, disruption is rapidly quenched.

As expected, the fall in test case failures is monotonic (Figure 6). However, root mean square (RMS) differences in values of affected subexpressions can rise and can fall (Figure 7), but RMS must be zero when values on all test cases are identical. For one example, fun 3, although the change is finite, it is immediately added to ±inf, so the ADD function yields ±inf regardless of the change, and fun 3's disruption does not propagate. In four more examples (fun 4, 6, 8 and 9),

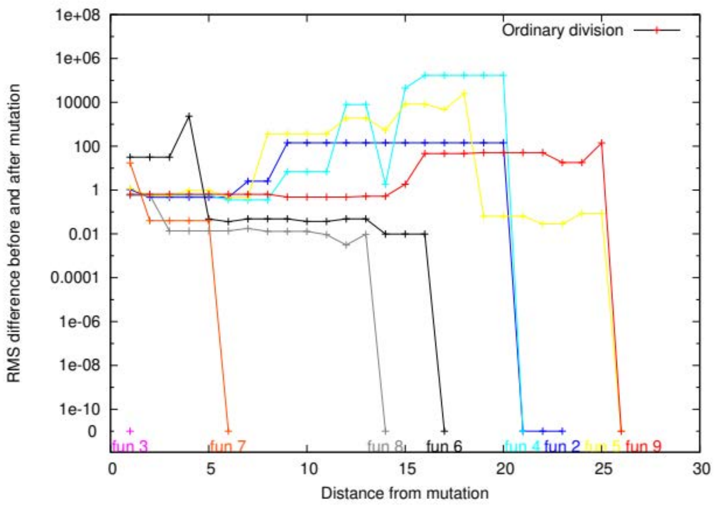


Figure 7. Change in impact, as measured by RMS difference on test cases, of eight changes with distance from disruption. In one change, fun 2, the floating-point difference on all test cases is zero (±0) after passing through 21 functions; however, two more functions are required to ensure exact binary equality. Note nonlinear scale.

the disruption spreads a little way before reaching an operation whose other argument is ±inf. In two others (fun 2 and 5), the disruption is suppressed effectively by a multiply by zero. The remaining change, fun 7, Section 7, is more complicated, but here too disruption is quickly suppressed.

6.2 Explaining Lack of Impact of Change 7

Table 1 shows that at 13 nodes, change 7 is one of the larger syntactic changes. Indeed, the orange line in Figure 2 shows it also produces a large change in behavior at the change site. (Change 7 replaces a constant by a large multiple of x.) See also Figure 8. At the point of disruption, all the test cases are different and the RMS difference can be estimated from the original constant b and the linear function $mx+a$ that replaces it (see caption of Figure 2). Setting $m=-29.9835$, $a=-0.24$ and $b=-0.725$ and approximating the sum over n test cases by an integral over -1 to $+1$ gives:

$$\begin{aligned}
\mathrm{RMS}_{\text{fun }7} &= \sqrt{\frac{1}{n}\sum_{n}(-29.9835x_i - 0.24 - -0.725)^2} \\
&\approx \sqrt{\frac{1}{2}\int_{-1}^{+1}(mx + a - b)^2} \\
&= \sqrt{\frac{1}{2}\int_{-1}^{+1} m^2x^2 + 2m(a-b)x + (a-b)^2} \\
&= \sqrt{\frac{1}{2}\left[m^2x^3 \,/\, 3 + (a-b)^2x\right]_{-1}^{+1}} \\
&= \sqrt{\frac{1}{2}\left(2m^2 \,/\, 3 + 2(a-b)^2\right)} \\
&= \sqrt{m^2 \,/\, 3 + (a-b)^2} \\
&= 17.3178.
\end{aligned} \tag{1}$$

This compares well with the actual value 17.335 obtained over a finite set of test cases (first orange line in Figure 7).

The next function up the tree toward the root node is an addition (Figure 8). Since this is linear, neither the number of test cases nor the RMS difference changes (see first orange line in Figures 6 and 7). Next is a multiply, whose other argument is $-0.003\,x$. This means for $x = 0$, the multiply node both before and after the change gives zero. This means that for the test case $x = 0$, the whole expression will give the same value before and after the change. Also the RMS difference at the multiply node is dramatically reduced (very approximately by a ratio of 0.003, Figure 7). The next three functions are linear and do not change either the number of test cases whose evaluation at each of the three functions is different before and after the change or the RMS difference. Immediately above them is a subtraction, whose other subtree is large and contains several divisions. It never gives finite values. (In fact, on these test cases it evaluates to ±inf or -nan.) Therefore so too does the subtraction. Notice that this particular node (only six levels above change 7) gives identical answers before and after the change and therefore immediately suppresses the remaining disruption caused by change 7.

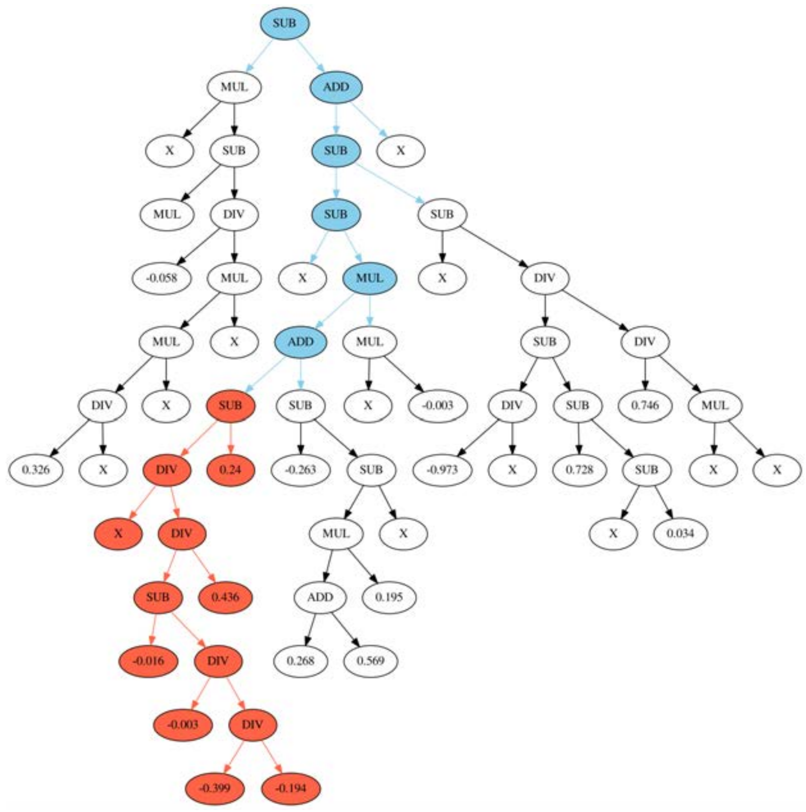


Figure 8. Fragment of change 7 with unprotected division. (Shown in full in Figure 5.) The original constant -0.725 is replaced by the subexpression (SUB (DIV X (DIV (SUB -0.016 (DIV -0.003 (DIV -0.399 -0.194))) 0.436)) 0.24) in red (here DIV is ordinary unprotected division). The blue nodes show operations in the original expression where their value on at least one test case is different before and after change 7. The white nodes indicate the evaluation is unchanged. The value of the new subexpression is given its topmost node (the topmost red SUB), and plotted as a function of x as the near-vertical line in Figure 2. Section 6.2 explains why disruption stops after six blue nodes, and so the red change makes no externally visible difference.

7. Experiment II: Protected Division

The problem of division by zero is well known in GP and is typically treated by "protecting" division [1, 2]. When the denominator is zero, the divide (DIV) operator is defined to return 1.0. We repeat the experiments in Section 6 with ordinary division replaced by protected division.

Contrasting Figures 4 and 9, we can see this simple change makes a dramatic difference in the values calculated by large expressions. However, although it takes the now infinity-free expressions longer to suppress the changes, passing the disruption through the arithmetic expression continues to suppress it. Contrasting Figures 6 and 7 with Figures 10 and 11 shows the average number of levels needed to totally hide a change has increased from 16 to 60.

Figure 5 shows results for both unprotected and protected division. Figure 5 shows the quick suppression of changes without protection against divide by zero as blue nodes, and with protected division as blue followed by green nodes.

In four changes (fun 2, 3, 7 and 8), the disruption is eventually caught by encountering a zero as the other argument. In three changes (fun 4, 5 and 6), the test cases are disabled progressively. This is also true of the last one, fun 9, but this is the only example where the disruption is not totally suppressed within the expression.

Figure 10 shows that indeed the number of test cases that are disrupted can remain the same or fall but can never rise. While Figure 11 shows, as expected, RMS differences can rise and fall.

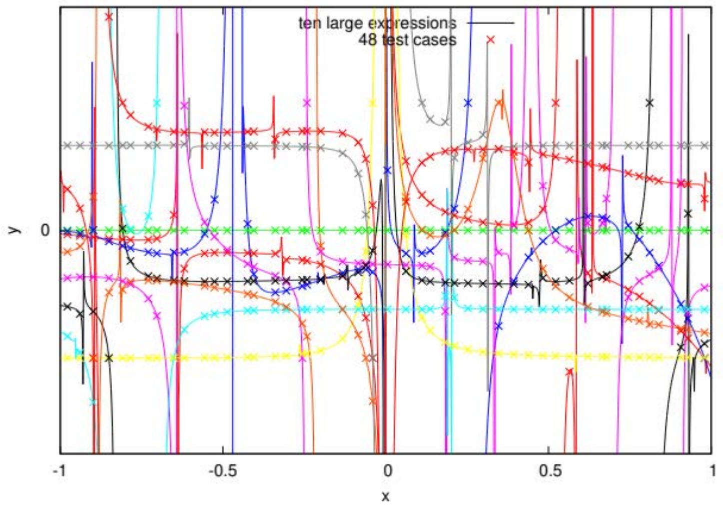


Figure 9. Ten large floating-point functions. Same expressions as Figure 4, except divide by zero is protected (Section 7). Vertical axis has been linearly rescaled to plot very different output ranges on the same axis. Here all outputs are finite. Tree 1 always gives 0 (green) but the other nine vary widely. (Note despite protecting divide by zero, rational functions often have discontinuities.)

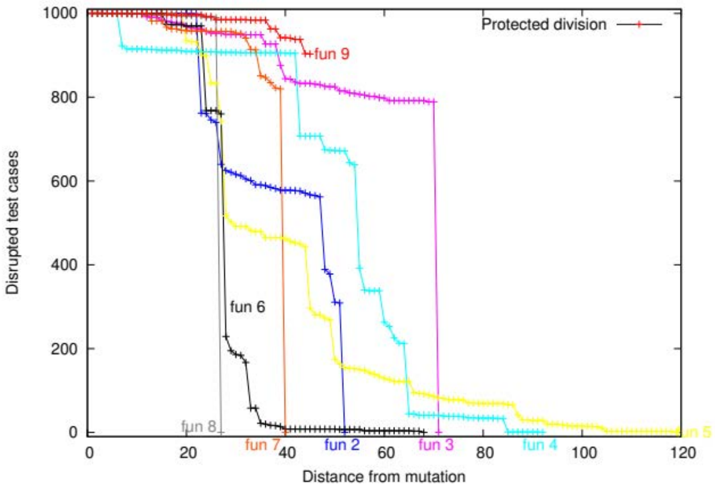


Figure 10. Fall in impact of eight changes with distance from disruption (functions 0 and 1 cause no disruption as both replace x with x and so are not plotted). As expected, the fall in test case failures is monotonic. Only fun 9 does not reach zero. Colors are the same as in Figure 2 and others.

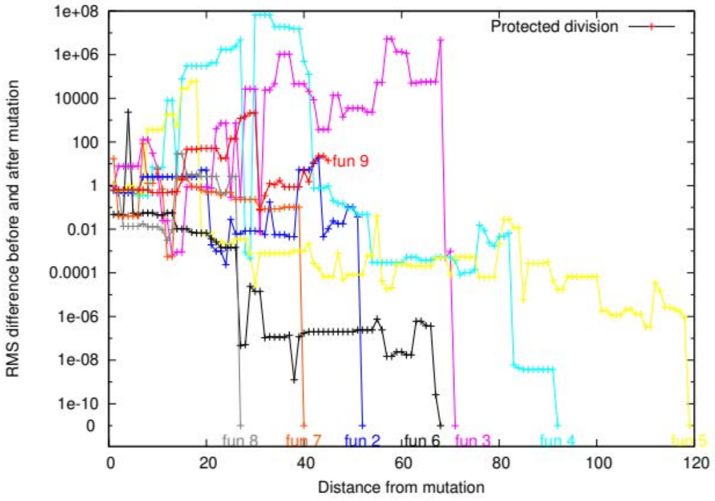


Figure 11. Change in impact, as measured by root mean squared difference on test cases, of eight changes with distance from disruption. Note nonlinear scale.

With the less dissipative functions, one change (9) is now visible on some test cases externally. This is because change 9 is not deeply nested inside its expression. (Table 1 shows it is only at depth 46.) Instead, although the change is suppressed on some test cases, there is not enough distance between the change location and the outside of the expression to suppress them all.

7.1 Explaining Lack of Impact of Change 4 with Protected Division

We choose change 4 as a second example to explain in detail because it cannot be explained as simply reaching an infinity or a multiply by zero and because its disruption passes through one of the longer chains of operations before being lost entirely. Figures 10 and 11 plot the gradual loss of disruption of change 4 and show on these test cases it is totally suppressed after 92 floating-point arithmetic operations. However, we shall not discuss in detail each of these 92 but concentrate on the first few, which hide change 4 on 86 test cases (Figure 12).

Change 4 is to replace the input x with the constant -0.052 (Table 1). In Figure 2, the original subexpression (x) is plotted along the diagonal, which makes it hard to see, as four other lines are also plotted along the diagonal. The replacement -0.052 is plotted in light blue with crosses. See also "fun 4" on right-hand margin.

The first operation involving change 4 is to replace $0.927-x$ with $0.927--0.052$. Clearly, for the test case where x is -0.052, these two expressions will produce the same answer. Not only that, but this unchanged value spreads through the whole expression like before the change. This means on test cases where x is -0.052, the whole expression (with 25 001 components) produces the same value as it did before the change. Thus at the lowest blue SUB node in Figure 12, the number of test cases that are different before and after the change is one less than the total number of test cases.

Since change 4 (like change 7) involved interchanging a linear function and a constant, we can reuse equation (1) to approximate the root mean square difference between the new expression and the original expression at the change site. Setting $m=1$, $a=0$ and $b=-0.052$ gives 0.579687, which is within 0.1% of the actual RMS.

The next four operations (ADD, ADD, ADD and SUB) are linear and do not change either the number of test cases where the new expression differs or the RMS value of the difference over the test cases.

Next is a division, in which x is divided by the value calculated by the previous SUB (five levels above the change). Neither before nor after the change does this subtraction produce zero; therefore for the test case where x is zero, both before and after the change, $x/..$ is also zero. That is, the number of test cases that are different falls by one.

Also note that, due to the nonlinear nature of DIV, RMS above and below the DIV operation is different.

The next operation is (SUB-0.175..); see Figure 12. Notice this linear operation reduces the number of test cases that differ by a further 76. When we study the internal traces of the original and the new expressions at this subtraction (Figure 13), we can see that they are similar. Indeed, within floating-point precision, at 78 test points they are identical. Notice again since subtraction is a linear operation, the RMS difference is unchanged.

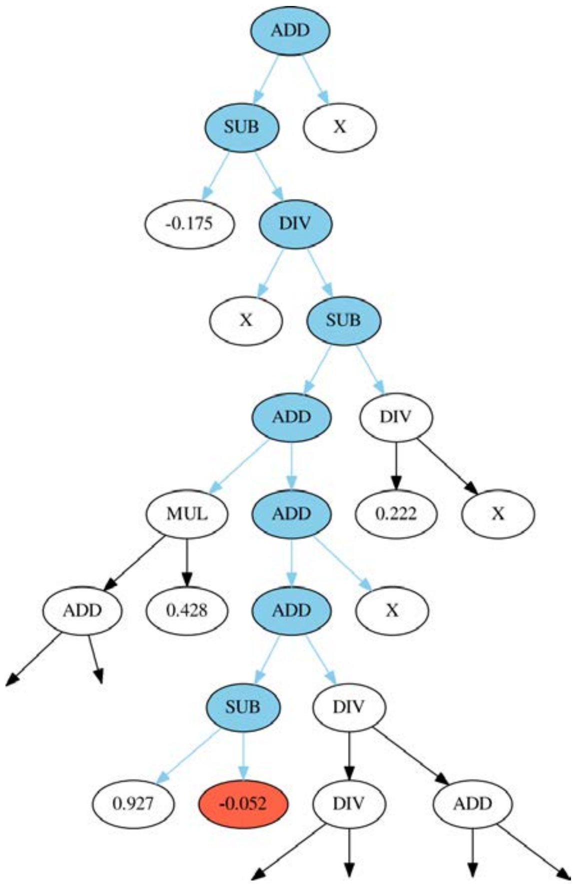


Figure 12. Fragment of change 4 with protected division. The original leaf x is replaced by the constant -0.052 in red. The blue nodes show the first few operations in the original expression where their value on at least one test case is different before and after change 4. (Here DIV is protected division.) White nodes show a fragment of unchanged large expression, shown in full in Figure 5. Section 7.1 explains the gradual dissipation of disruption. After 92 blue nodes (lowest 8 shown), change 4 makes no visible difference.

The next operation (ADD .. x) is again linear (so RMS is unchanged). However, adding x nudges a further eight test points into being identical. Figure 14 zooms in on the 86 test points where the changed and unchanged subexpressions give identical values.

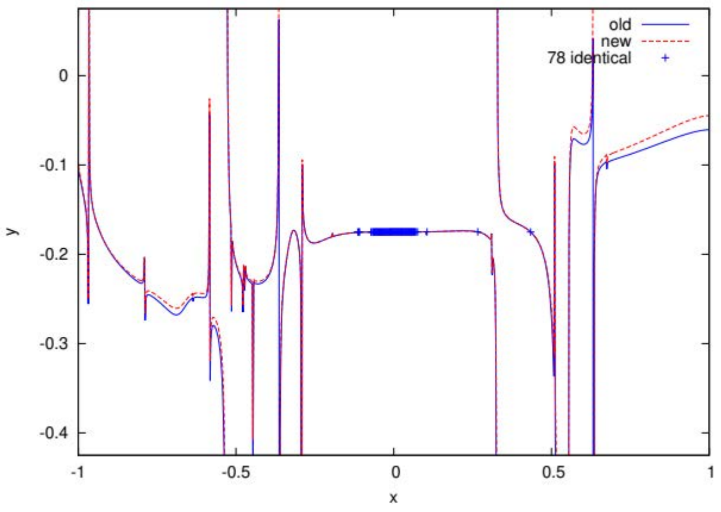


Figure 13. Impact of change 4 with protected division at distance seven. The new functionality (dashed line) closely follows the original. Indeed, at 78 points (+) they are identical.

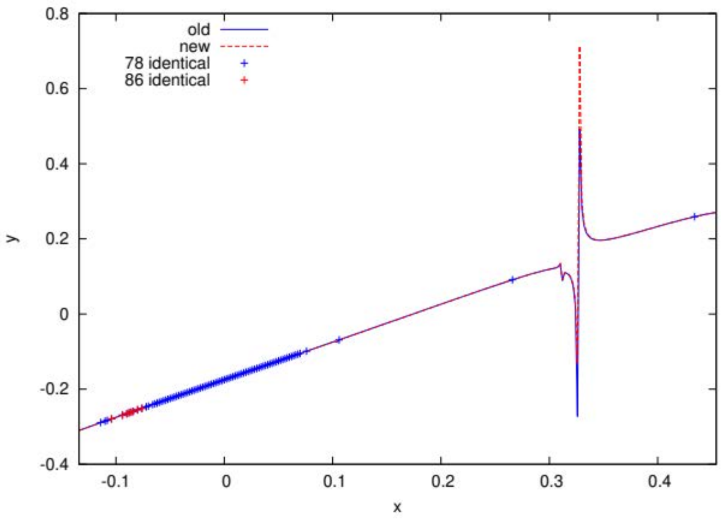


Figure 14. Impact of change 4 with protected division at distance eight (i.e., adding x to value produced by SUB, Figure 13). The new functionality (dashed line) closely follows the original. At 86 points (+) they are identical. This is eight more test points (red +) than the previous subtraction.

As the disruption caused by replacing x by -0.052 spreads throughout the large expression, it continues to loose potency, so that after being transformed by 92 arithmetic operations, it is not visible at all on any of the test points.

8. Conclusion

Large floating-point arithmetic expressions are robust in the sense that the value they calculate tends to be the same after a small syntactic change as it was before.

With modern floating point, large arithmetic expressions are dominated by nonfinite values such as ±inf and ±nan, which are readily generated by division by zero. Small changes to such expressions are easily lost due to arithmetic on nonfinite values and due to operations like multiply by zero, whose output does not depend on their other input.

When care is taken to avoid nonfinite values, binary operations like multiply by zero or division with zero as a numerator continue to suppress information about changes to their other argument.

These special values (0, ±inf and ±nan) are easy to explain and affect many arithmetic values, but they are special examples of a general trend for slightly different arithmetical expressions to calculate the same value (within common machine precision). This was observed in all our experiments (e.g., Figures 6 and 10). In one example, where the disruption was the closest to the root, the change was partially visible externally. In all the other examples, the disruption was buried so deeply that it had to pass through enough intermediate floating-point arithmetic operations that it was not observed externally at all.

Acknowledgments

I would like to thank Justyna Petke, EPSRC grant EP/P005888/1 (gow.epsrc.ukri.org/NGBOViewGrant.aspx?GrantRef=EP/P005888/1) and the Meta OOPS project.

References

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

[2] R. Poli, W. B. Langdon and N. F. McPhee (with contributions by J. R. Koza), *A Field Guide to Genetic Programming*, 2022. (Aug 26, 2022) www.gp-field-guide.org.uk.

[3] W. B. Langdon and J. Petke, "Software Is Not Fragile," in *First Complex Systems Digital Campus E-conference (CS-DC'15)* (P. Bourgine, P. Collet and P. Parrend, eds.), Cham: Springer, 2015 pp. 203–211. doi:10.1007/978-3-319-45901-1_ 24.

[4] J. Petke, B. Alexander, E. T. Barr, A. E. I. Brownlee, M. Wagner and D. R. White, "A Survey of Genetic Improvement Search Spaces," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO 2019)*, Prague, Czech Republic (B. Alexander, S. O. Haraldsson, M. Wagner and J. R. Woodward, eds.), New York: Association for Computing Machinery, 2019 pp. 1715–1721. doi:10.1145/3319619.3326870.

[5] G. Jahangirova, *Oracle Assessment, Improvement and Placement*, Ph.D. thesis, Department of Computer Science, University College London, UK, April 16, 2019. discovery.ucl.ac.uk/id/eprint/10072699.

[6] R. E. Lenski, M. J. Wiser, N. Ribeck, et al., "Sustained Fitness Gains and Variability in Fitness Trajectories in the Long-Term Evolution Experiment with *Escherichia coli*," *Proceedings of the Royal Society B: Biological Sciences*, **282**(1821), 2015 20152292. doi:10.1098/rspb.2015.2292.

[7] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz and S. Yoo, "The Oracle Problem in Software Testing: A Survey," *IEEE Transactions on Software Engineering*, **41**(5), 2015 pp. 507–525. doi:10.1109/TSE.2014.2372785.

[8] T. Y. Chen, "Metamorphic Testing: A Simple Approach to Alleviate the Oracle Problem," in *2010 Fifth IEEE International Symposium on Service Oriented System Engineering (SOSE '10)*, Nanjing, China, 2010 (X. Bai and Y. Li, eds.), Piscataway, NJ: IEEE, 2010 pp. 1–2. doi:10.1109/SOSE.2010.31.

[9] W. B. Langdon and O. Krauss, "Genetic Improvement of Data for Maths Functions," *ACM Transactions on Evolutionary Learning and Optimization*, **1**(2), 2021 pp. 1–30. doi:10.1145/3461016.

[10] W. B. Langdon, "Long-Term Evolution of Genetic Programming Populations," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO '17)*, Berlin, Germany 2017, New York: Association for Computing Machinery, 2017 pp. 235–236. doi:10.1145/3067695.3075965.

[11] W. B. Langdon and W. Banzhaf, "Long-Term Evolution Experiment with Genetic Programming," *Artificial Life*, **28**(2), 2022 pp. 173–204. doi:10.1162/artl_a_00360.

[12] W. B. Langdon, "Genetic Programming Convergence," *Genetic Programming and Evolvable Machines*, **23**(1), 2022 pp. 71–104. doi:10.1007/s10710-021-09405-9.

[13] W. B. Langdon, “Evolving Open Complexity,” *SIGEVOlution Newsletter of the ACM Special Interest Group on Genetic and Evolutionary Computation*, **15**(1), 2022 pp. 1–4. doi:10.1145/3532942.3532945.

[14] S. Wolfram, *A New Kind of Science*, Champaign, IL: Wolfram Media, Inc. 2002.

[15] T. Rowland. “Computational Irreducibility” from MathWorld—A Wolfram Web resource. mathworld.wolfram.com/ComputationalIrreducibility.html.

[16] E. W. Weisstein. “BBP Formula” from MathWorld—A Wolfram Web resource. mathworld.wolfram.com/BBPFormula.html.

[17] M. Alekseyev. “Height (Maximum Absolute Value of Coefficients) of the n^{th} Cyclotomic Polynomial.” The On-Line Encyclopedia of Integer Sequences. (Aug 26, 2022) oeis.org/A160338.

[18] H. Zenil, N. A. Kiani, A. A. Zea and J. Tegner, “Causal Deconvolution by Algorithmic Generative Models,” *Nature Machine Intelligence*, **1**(1), 2019 pp. 58–66. doi:10.1038/s42256-018-0005-0.

[19] H. Zenil, N. A. Kiani, F. Marabita, Y. Deng, S. Elias, A. Schmidt, G. Ball and J. Tegner, “An Algorithmic Information Calculus for Causal Discovery and Reprogramming Systems,” *iScience*, **19**, 2019 pp. 1160–1172. doi:10.1016/j.isci.2019.07.043.

[20] H. Zenil, N. A. Kiani, F. S. Abrahao and J. N. Tegner, “Algorithmic Information Dynamics,” *Scholarpedia*, **15**(7), 2020 53143, revision #195807. doi:10.4249/scholarpedia.53143.

[21] W. B. Langdon, *Fast Generation of Big Random Binary Trees*, Technical Report RN/20/01, Computer Science, University College, London, Gower Street, London, UK, 13 January 2020. arxiv.org/abs/2001.04505.

[22] P. Flajolet and A. Oldyzko, “The Average Height of Binary Trees and Other Simple Trees,” *Journal of Computer and System Sciences*, **25**(2), 1982 pp. 171–213. doi:10.1016/0022-0000(82)90004-6.

[23] W. B. Langdon, “Incremental Evaluation in Genetic Programming,” in *Genetic Programming (EuroGP 2021)*, Virtual Event, 7–9 April 2021 (T. Hu, N. Lourenco and E. Medvet, eds.), Cham: Springer, 2021 pp. 229–246. doi:10.1007/978-3-030-72812-0_15.

[24] A. Singleton, “Genetic Programming with C++,” *BYTE*, 1994 pp. 171–176.