Genetic Improvement of Data Gives Division free Division

William B. Langdon

Oliver Krauss
CREST, Computer Science, UCL, London, WC1E 6BT, UK
Johannes Kepler University Linz, Austria AIST, University of Applied Sciences Upper Austria

***Abstract*—Modern society is dependent on software. Yet software production is labour intensive. While most software automation research concentrates on programs' code, we have started investigating if genetic improvement can assist developers by automating aspects of the maintenance of parameters embedded in software. We extend recent GI work on optimising compile time constants to give new functionality and fully describe the transformation of a C library square root function into the reciprocal function. Multiplying by the reciprocal allows division without requiring the hardware to support division. The evolution and indeed the evolved function are both surprisingly fast.**

I. Conventional Division is Expensive

Considering just one core, on a modern 3.60 GHz desktop on average double precision multiplication takes in the region of a nanosecond, whereas double precision division takes about 4.0 times as long. For some systems which do not have floating point division in hardware, e.g. MMM [1], the ratio may be bigger. Even in some cases with hardware division, such as an 2004 ARM Vector Floating-Point coprocessor [2, VFP 1-19] the ratio is 14.5. Minimal systems, such as for internet-of-things (IoT) or mote computing may not have the transistors or the power for conventional hardware division. In contrast other machines dedicated to computational performance, such as GPUs, may have a variety of implementations of multiplication and division and even include specialised operations such as reciprocal (invert, rcp or in double precision drcp).

We use genetic improvement operating on *data* to show search can adapt existing fluid embedded constants to repurpose existing open source C code to give a software double precision implementation of reciprocal (drcp). Our table driven drcp takes only 4 Kbytes (512×4bytes, which could be burnt into read-only memory (ROM) and as such is well within the reach of many mote processors.

Not only can artificial evolution do this, but, when used inconjunction with multiply, we get a table driven software implementation of division which on our desktop is only marginally slower than native 64 bit division. (To be exact 68% slower, and on an older E5462 2.80GHz CPU the difference is even smaller at 50%.) While this is not competitive on a full blown modern desktop CPU, it may be useful for systems where the ratio of performance of division to multiplication is more than 8. E.g., the ARM1176JZF-S [2], where double precision multiply is 14.5 times faster than division.

The next section gives the background of software engineering maintenance, particularly using Artificial Intelligence (AI) techniques, such as evolutionary computing (EC). Section III gives a brief introduction to iteratively finding a root of a mathematical equation using Sir Isaac Newton and Joseph Raphson's iterative solver. Section IV describes a GNU C mathematics library routine, sqrt, which uses Newton-Raphson and how we use EC to evolve the data within it to give a new double precision reciprocal function, drcp, which does not use double precision division and can be used to replace it. The discussion (Section V) considers if our GI division is correct and useful and possible future work. Finally, in Section VI we conclude that for some low resource computers the GI approach may help and that we have demonstrated AI tools like CMA-ES are opening up new approaches to automating software maintenance.

II. Background: AI for Software Maintenance

Although computing is without doubt the success story of the second half of the twentieth century, at the beginning of the third millennium we are faced with an IT industry which remains labour intensive but not in the manufacture of the things but in looking after intangible IP (intellectual property), principally software. The lifetime cost of solid things, i.e. hardware, has fallen exponentially [3]. However, there has been no such dramatic change in the cost of intangibles. Whilst production of hardware has been automated, software has not been automated and remains labour-intensive. These differences between computers and the software that runs on them has lead to very different maintenance regimes.

Forty years ago there were a (relatively) small number of large computers and teams of technicians who performed regular "preventative maintenance" on them. Those days are long since past. Nowadays the number of computers is vast, and their components so tiny and interconnected that routine maintenance is no longer attempted. Instead whole computers (never mind components) are simply discarded when (part of them) fails. Contrast this with their software.

Initially, computer hardware was very diverse, demanding new software each time new hardware was commissioned. The advent of high-level languages and near monopolies in computer manufacture has lead to increasingly large volumes of software which can and has been reused. This led to "immortal software" which has long outlived the computers it was originally developed for. Surprisingly the apparent reduction in hardware and operating system environments has not lead to a similarly stable software industry. The economic pressure to be "first to market" has forced hardware innovation

to be concentrated in a few computer hardware companies with fixed upgrade tram lines leading to users being "locked in". However, for most software the race to be first continues to lead to great diversity in user programs.

The lack of automation in the software industry, online deployment and the economic necessity for software to be produced quickly, has led to the rise of continuous deployment where new software is inflicted on the user as quickly as possible. Innovations in software production have fed the race to be fast rather than increasing software quality. The lack of automation and the longevity of software have led to software maintenance becoming the dominant cost of computing.

Search based software engineering (SBSE) [4] uses AI tools to tackle software engineering problems. Increasingly SBSE is being used not just to find solutions to software problems, but to help software writers and maintainers by increasing the level of automation [5], [6]. Rather than generating totally new programs [7]–[9], we are now seeing AI being used to automatically fix bugs [6], [10], [11] and improve existing software [12]–[14]. Genetic Improvement (GI) [14] has been used to optimise run time [13], [15] energy [16]–[18] and memory [19] efficiency, automatically importing functionality from one program into another [20], [21], growing new functionality and grafting it into another [22]–[25] and indeed porting to new hardware [26]. GI on source code systems include GIN [27] (Java), PyGGI [28] (multiple) and GISMOE [29] (C). Although GI can be applied to byte code [30], assembler [31] and indeed machine code [32] mostly GI has been applied to program source code, with little on numbers embedded in software.

As well as external data to be processed, typically programs contain not just computer instructions, but also data. These may be `float` or `int` values or other. For example, the GNU C library contains more than a million integers, see Figure 1, also [33]). The numbers can be integral to the source code itself, but may also relate to the problem the program is solving, and as such may be subject to change just like the rest of the program's environment and so may need to be updated. The data maintenance problem has been recognized for a long time (Martin and Osborne, 1983 [34, Section 6.8]).

Although maintenance is the dominant cost of computing a recent survey [35] starts by saying "a relatively small amount [of SBSE research] is related to software maintenance", whilst de Freitas and de Souza [36] do not give a break down of the search based software engineering literature on software maintenance. Indeed there is little SBSE research on maintaining embedded numbers.

There is a little research on tuning of embedded parameters, Wu et al. [19]'s deep parameter (DPO) work being the first example. They used DPO to adjust a few parameters to reduce runtime and memory. However, unlike Wu et al. [19], we focus on adapting many numerical values to give better programs or indeed (as here) new functionality.

The ViennaRNA package [37] uses more than 50 000 free energy values. These came from measurements of RNA molecules. However, whilst the package has been used sci-

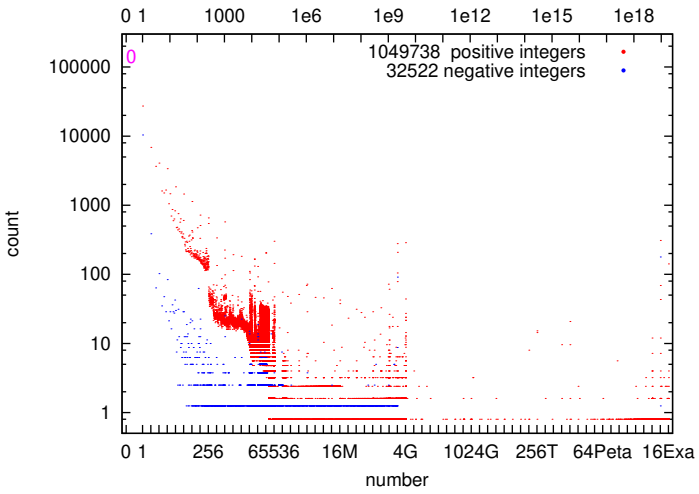


Fig. 1. Distribution of integer constants in the GNU C library version 2.30 released 1 August 2019 (including test suite). Note log scales. It contains 967 533 lines of C code, which contain in total 1 234 449 integer constants. Zero is the most common occurring in various formats a total of 152 189 times, followed by 1 (33 985) 2 (8 594) and -1 (8 324). Every integer between -50 and 40 956 occurs at least once. There are 118 386 distinct integer constants. (To avoid overlap, positive and negative values are slightly offset vertically.)

entific knowledge has moved on and various newer data values are available. Recently [38] we showed that genetic improvement can adapt these 50 000 `int` values. (The GI parameters are distributed with RNAfold version 2.4.5 onwards.) In [38] we used custom mutation and crossover operators. Since RNAfold was recompiled and tested, evolution took about five days (rather than a few seconds in these examples). Notice that there were no changes to the code. Only data were changed.

We showed that evolution could update thousands of embedded constants to give new functionality [38], [39]. In [39] we argued that the technique could be widely applied and have applied it to generating $\log_2$ [40] and $\frac{1}{\sqrt{x}}$ [41]. We now use it to evolve a double precision division operator without division. We provide double precision division, x/y, as $x \times (1/y)$, i.e. $x \times \text{drcp}(y)$. Where $\text{drcp}(x) = 1/x$ is the double precision reciprocal or invert function.

III. Newton-Raphson

Newton-Raphson is an iterative way of finding the roots (zero crossing points) for continuous differentiable functions, see Figure 2. Under ideal conditions it converges quadratically fast. Thus if we start with an 8 bit approximation, the next iteration is accurate to 16 bits, the second 32 bits and the third to 64 bits. Since double precision (see Figure 3) gives 52 bit accuracy, only three Newton-Raphson cycles are needed. Classically, each Newton-Raphson iteration includes testing to see if the new approximation is close enough and stopping when the error is small enough. For speed, in the GNU e_sqrt.c code, the test is omitted and it simply does three iterations and stops.

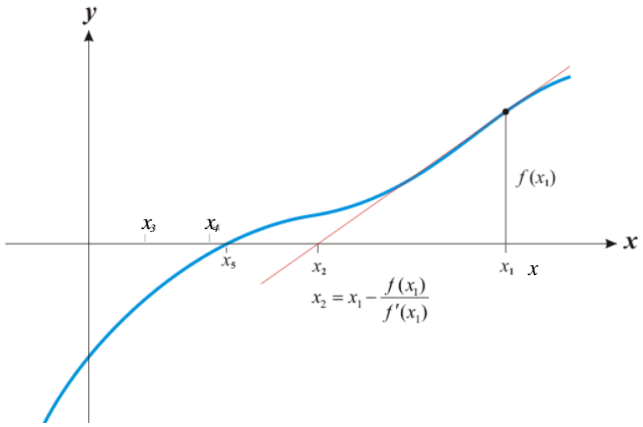


Fig. 2. First iteration of Newton-Raphson to approximate root $f(x)$=0 of a function, thick blue line, (Wikipedia). $f'(x_1)$ is the derivative of f at x_1 (thin red line). Following it gives x_2 where it crosses the horizontal line $y = 0$, $x_2 = x_1 - \frac{f(x_1)}{f'(x_1)}$. Here x_2 is closer than x_1 to the root. The next iteration starts at x_2 to give $x_3 = x_2 - \frac{f(x_2)}{f'(x_2)}$. In this example x_3 over shoots but x_4 is close and x_5 is almost exact.

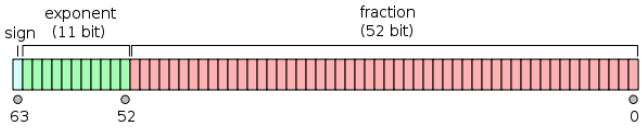


Fig. 3. IEEE 754 Double-precision floating-point format (Wikipedia).

Using Newton-Raphson to find $f(x) = 0$, see Figure 2. Guess an initial value for x, x_1. The initial error is:

$$\text{error} \quad = \quad f(x_1) - 0$$

The next estimate, x_2, is given by updating x_1 by the error divided by the gradient (derivative) of $f(x)$, $f'(x)$

$$\begin{aligned} x_2 &= x_1 - \text{error}/f'(x_1) \\ &= x_1 - \frac{f(x_1)}{f'(x_1)} \end{aligned}$$

Generalising, for the $(n+1)^{\text{th}}$ step

$$x_{n+1} \quad = \quad x_n - \frac{f(x_n)}{f'(x_n)}$$

The GNU C library sqrt code contains a table of 512 (2^9) values for x_1 (and another 512 holding initial values for the derivative, see next paragraph). The table is indexed by a 9 bit integer ($0 \ldots 511$) extracted using bit shifts and masks from the `double` format number, Figure 3. Each start point x_1 is accurate to within eight bits and so three iterations will give double precision accuracy. (Since initially only 8 bit precession is needed, the 512 pairs of initial values can be stored as `float` rather than `double`.)

In the case of sqrt $f(x) = x^2 - a$. Note when $f(x) = 0$ then $x = a^{\frac{1}{2}}$. To use multiplication instead of division, the GNU C library sqrt code keeps an estimate of $\frac{1}{f'(x)}$ which is updated at each Newton-Raphson iteration.

In our case, the reciprocal function, $f(x) = x^{-1} - a$ and so $f'(x) = -x^{-2}$:

$$\begin{aligned} \frac{f(x_n)}{f'(x_n)} &= -f(x_n)x_n^2 \\ &= -\left(x_n^{-1} - a\right) x_n^2 \\ &= -x_n + a\, x_n^2 \end{aligned} \tag{1}$$

Since x^2 is easily calculated, for drcp (unlike sqrt) we do not maintain an estimate for the reciprocal of the derivative. Therefore this part of `e_sqrt.c` was removed and x^2 was used instead (see Section IV-A).

IV. Evolving drcp from free GNU PowerPC sqrt

We use an existing table driven implementation of the square root function (Figure 4 left) and use genetic improvement to evolve the reciprocal function (Figure 4 right). This is achieved by mutating the constant values in the chosen code for square root.

The GNU C library (release 31 Jan 2019 glibc-2.29) was downloaded[1]. It contains multiple implementations of the square root function (sqrt). As before [39], we selected `sysdeps/powerpc/fpu` from the PowerPC implementations as it uses table lookup [42] (Figure 5). Again we adapted the GNU open source C code by hand and used CMA-ES to evolve the fluid literals supplied by GNU for the square root function so that the code now calculates drcp.

A. Manual changes

A few small code modifications are needed before running evolution on the data table (contrast Figures 5 and 6).

- Like sqrt, negative numbers are caught before entering the main code. However, unlike sqrt, an error is not raised but instead $-x$ is passed to the main code and its result is negated. Thus, as with sqrt, the main code does not deal with negative numbers.
- The construction of the nine bit indexing operation is essentially unchanged, but it must take into account that the table contains 512 floats not 512 pairs of floats (Figure 6).
- The code to maintain the estimate of the reciprocal of the derivative can be commented out.
- The new formula (Eq. 1) for the Newton-Raphson step is used (three times).
- The GNU sqrt code deals with the exponent separately from the fractional part (Figure 3). To take the square root of the exponent, it is divided by two using a 1 bit shift right (left part Figure 5). For x^{-1} the exponent part must be negated. Since the IEEE 754 standard uses 11 unsigned bits to represent the exponent, the new code subtracts it from the mid point ($1023 = 2^{11}/2 - 1$) giving the exponent of the result (left part Figure 6). That is, the new code replaces masks and shift by masking and subtraction.

[1] https://ftp.gnu.org/gnu/glibc/glibc-2.29.tar.gz

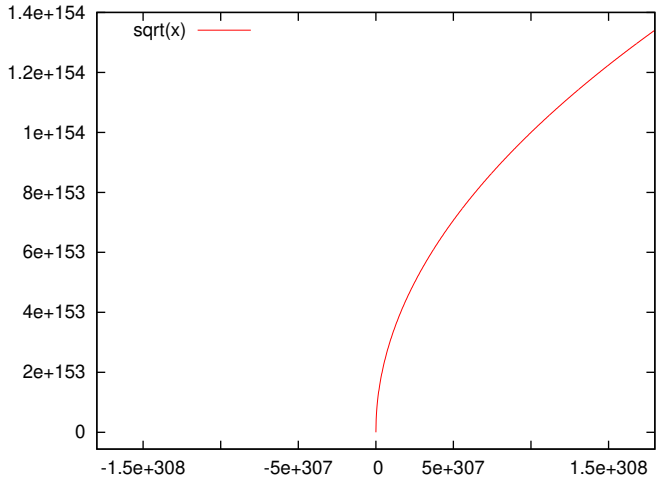
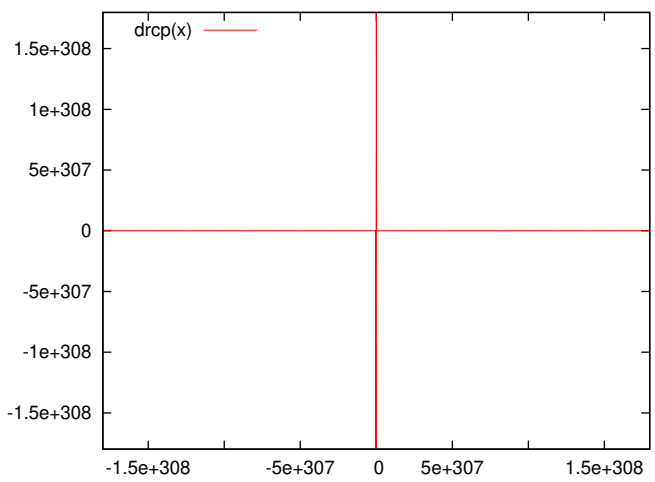


Fig. 4. Left: Double precision square root



Right: 1/x Double precision reciprocal

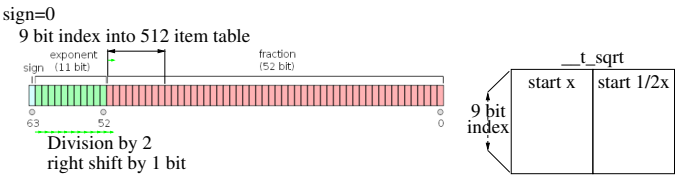


Fig. 5. Left: e_sqrt.c uses right shift of exponent of positive double to a) divide exponent by two and b) merge least significant bit with top 8 bits of fractional part to give nine bit index. Right: index used with `float` __t_table containing 512 x_1 and $1/2x_1$ pairs of initial values for Newton-Raphson iterative solution of $\sqrt{x}$.

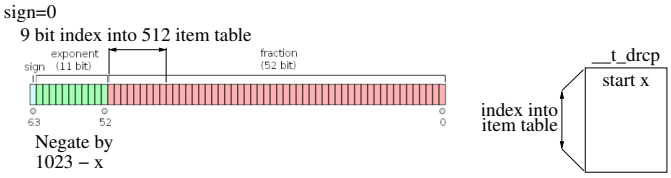


Fig. 6. GI drcp. Left: to negate the exponent of positive double (11 bit twos complement integer), drcp subtracts it from 1023. drcp uses the top 9 bits of fractional part give a nine bit index. Right: index used with `float` __t_drcp of initial values for Newton-Raphson iterative solution of $\frac{1}{x}$.

- The sqrt code deals with denormalised numbers (i.e. when the exponent part is zero, $x < 2^{-1023}$, see Figure 3) by multiplying by a large number and recursively calling itself and then adjusting the returned value appropriately. Except for using 2^{54}, the new drcp is identical. It multiplies the tiny value x by 2^{54}. The drcp code is recursively called with the new (now normalised double precision value). The output will be 2^{-54} times too small and so the correct final value is obtained by multiplying by 2^{54}. I.e. the e_sqrt.c macros `two108`$= 2^{108}$ and `twom54`$= 2^{-54}$ are both replaced by `two54`$= 2^{54}$.

B. Automatic changes to data table using CMA-ES

The GNU __t_sqrt table contains 512 pairs of floats. The first of each pair was used as the starting points when evolving the 512 floats in the new table, see horizontal axis Figure 9. (The float values found by CMA-ES are shown by the vertical axis of Figure 9, also Figure 10.)

The Covariance Matrix Adaptation Evolution Strategy algorithm (CMA-ES [43]) was downloaded from https://github.com/cma-es/c-cmaes/archive/master.zip It was set up to fill the table of floats one at a time. In all cases, the initial mutation step size used by CMA-ES was set to 3.0 times the standard deviation calculated from the 512 x_1 values in __t_sqrt.

1) CMA-ES parameters: The CMA-ES defaults (`cmaes_initials.par`) are used, except for: the problem size (N=1), the initial values and mutation sizes are loaded from __t_sqrt (see previous section), and stopFitness, stopTolFun, stopTolFunHist and stopTolX, which control run termination, were set to zero to ensure CMA-ES tries to get a perfect fitness (see next section).

2) Fitness function: Each time CMA-ES proposes a double value (N=1), it is converted into a float and loaded into the table at the location that CMA-ES is currently trying to optimise. The fitness function uses three fixed test double values in the range 1.0 to 2.0. These are: the lowest value for the table entry, the mid point and the top most value. The invert function drcp that CMA-ES is trying to create is called (using the updated table) for each and a sub-fitness value calculated with each of the three returned doubles. The sub-fitnesses are combined by adding them.

Sub-fitness is calculated by taking CMA-ES's drcp output and inverting it (using $1.0/x$). If the evolved value was correct, the answer would be the same as the test input value. (Effectively the fitness function is using metamorphic testing, which avoids having a test oracle which knows in advance the desired answer for every fitness test case.) Sub-fitness is based on the absolute difference between these. If they are the same or very close, the sub-fitness is 0. We define "very close" to mean the difference is less than either the difference calculated when inverting a number very slightly smaller than CMA-ES's drcp's output or when inverting a number very slightly bigger.

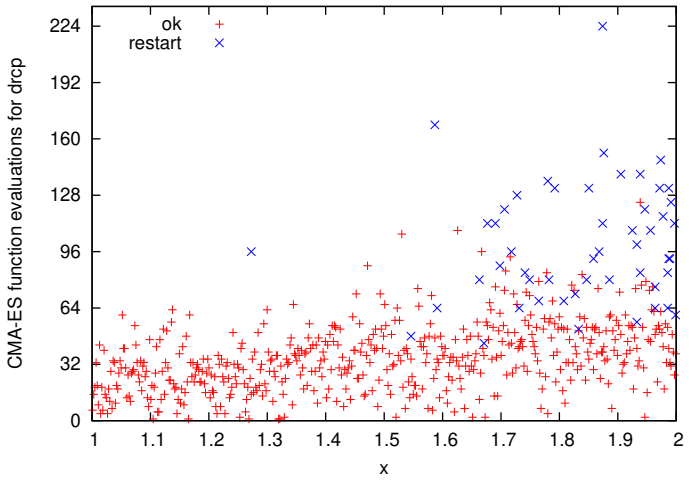


Fig. 7. CMA-ES is very good at finding 512 new start points for x^{-1} when starting with data for $x^{\frac{1}{2}}$. (Mean 45.6 fitness evaluation.) All but 1 of the 51 of the runs which were restarted ($\times$) are for $x > 1.5$ indicating some correlation with change to initial seed value (Figure 9).

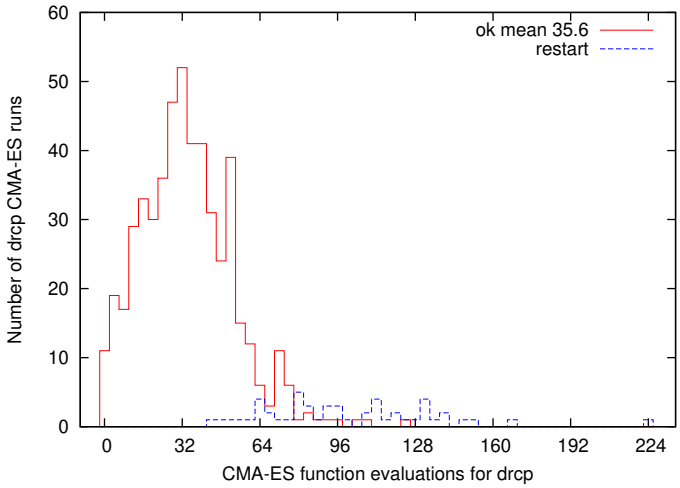


Fig. 8. Histogram (bin size 4) of number fitness evaluations per run for successful runs (solid line) and for 51 runs which did not find an acceptable solution immediately (blue dashed line). Data as Figure 7.

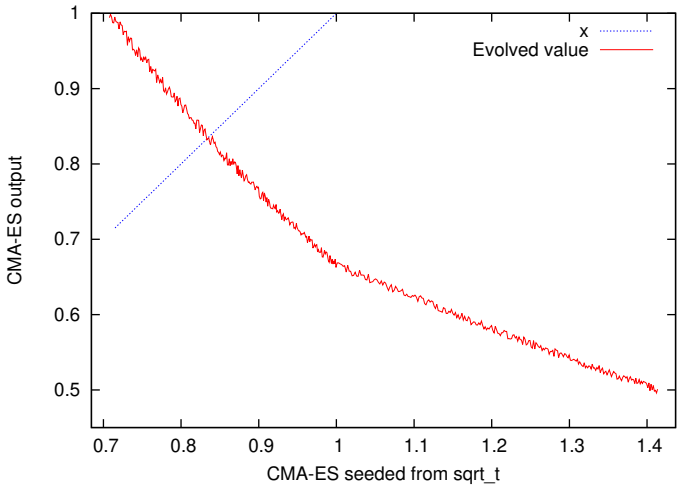


Fig. 9. Evolved change from sqrt table values (horizontal axis) to corresponding inv table value (vertical axis). 512 successful CMA-ES runs. (Diagonal blue line shows no change, y=x)

"Slightly" meaning to the best double precision accuracy, i.e. multiplied or divided by (1+DBL_EPSILON) = $(1+2^{-52})$.

If the output from the evolved drcp is not close enough, the sub-fitness is positive. When drcp is working well the differences are very small, therefore they are re-scaled for CMA-ES (although this may not be essential [44]). If the absolute difference is less than one, its log is taken, otherwise the absolute value is used. However, in both cases, to prevent the sub-fitness being negative, log of the smallest feasible non-zero difference DBL_EPSILON is subtracted.

CMA-ES will stop when the fitness is zero, i.e. the errors on all three test points are close enough to zero.

3) Restart Strategy: If CMA-ES fails to find a value for which all three test cases are ok, it is run again with the same initial starting position and mutation size, but a new pseudo random number seed. In 467 cases CMA-ES found a suitable value in one run, but in 39 of 512 cases it was run twice, and in 6 cases three CMA-ES runs were needed.

It took 6 seconds to run CMA-ES 563 times on one core of a 3.60GHz i7-4790 Intel desktop computer. The search effort is given in Figures 7 and 8. The values input to CMA-ES and those output by it are given in Figures 9 and 10.

C. Testing the evolved drcp function

The glibc-2.29 powerPC IEEE754 table-based double sqrt function claims to produce answers within one bit of the correct solution. Our drcp also achieved this. On 1 543 tests of large integers ($\approx 10^{16}$) designed to test each of the 512 bins three times (min, max and a randomly chosen point) the largest discrepancy between 1/drcp(x) and x was two, i.e. a maximum fractional error $1.9\ 10^{-16} \approx$ DBL_EPSILON.

The evolved drcp was also tested with 5 120 random numbers uniformly distributed between 1 and 2 (the largest deviation was two[2]), 5 120 random scientific notation numbers and 5 120 random 64 bit patterns. Half the random scientific notation numbers were negative and half positive. In absolute value, half were smaller than one and half larger. The exponent was chosen uniformly at random from the range 0 to $|308|$.

In one case a random 64 bit pattern corresponded to NAN (Not-A-Number) and drcp correctly returned NAN. In five cases random 64 bit pattern corresponded to numbers either bigger than 2^{1023} or smaller than -2^{1023}. For these drcp correctly returned 0 (or -0). In most cases drcp returned a double, which when inverted was its input or within one bit of it. Barring the six special binary patterns, the maximum deviation was 2, i.e. $4.44\ 10^{-16}$ as a fraction.

[2]2 at the least significant part of IEEE754 double precision corresponds to $4.44\ 10^{-16}$.

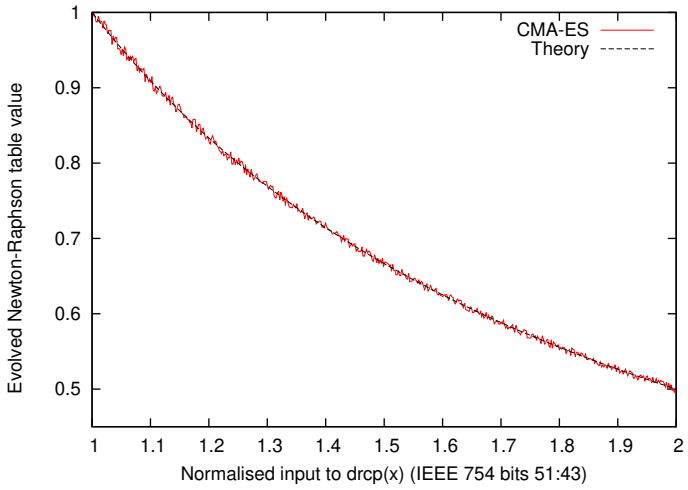


Fig. 10. Vertical axis same values as Figure 9. I.e. 512 GI table values for drcp x^{-1}. They are shown plotted against normalised x input to $1/x$ on the horizontal axis.

V. Discussion

A. Sufficient Testing?

Although it is well-known that testing cannot prove correctness [45], in keeping with the IT industry as a whole (indeed the GNU developers themselves, see Section V-C), we have sought to demonstrate our evolved implementation using testing (see Section IV-C). Excluding tests for exceptions such as denormalised numbers and range errors, the code is straight through with no loops or branches and drcp (in the range of interest, 1.0 to 2.0) is monotonic and smooth. We have shown this main code yields correct double precision answers thousands of times, including covering all 512 data bins multiple times, including their edges. (Indeed a proof, following Markstein [42], might be possible.) We can thus be reasonably confident of the GI code in normal operation.

Although the testing has covered some special cases, it is noticeable that the GNU mathematics library developers have included almost as many tests for exceptions as for normal cases. We have been concerned primarily with normal operation and not attempted to use the glibc exception handling code. Therefore if our evolved drcp were to be included in glibc they might well want to satisfy themselves that non numeric inputs such as nan, +inf and -inf are also dealt with correctly. Similarly additional testing might be used when dealing with non-normalised numbers, cf. Section IV-A, especially as this is the only instance of recursive code.

B. Originality, Utility and Scope for Disruption of SE

As the literature review in Section II makes clear this is an under explored area and yet (particularly, as here, where Evolutionary Computing (EC) can be applied to data without recompilation) EC can produce useful results with a run time of a few seconds. By working with software maintainers EC based AI data maintenance tools could make a significant dent in the software maintenance mountain.

As mentioned in Section I, our evolved double precision division, $x/y = x \times \text{drcp}(y)$ is potentially competitive on processors where division is slow. In particular, if division is slower than multiplication by a ratio of more than 8, multiplication by the evolved double precision reciprocal, drcp, will be faster than division. For example, there are some processors (see Section I) where the time to do double precision division is more than $14\times$ the time to do double precision multiplication, e.g. ARM1176JZF-S [2] On such hardware $x \times \text{drcp}(y)$ would be faster than x/y.

As before [39] we have only updated the read-only data table. Comments in the GNU PowerPC C source code make the point that a skilled software designer chose the interleaving of the instructions to maximise performance on the PowerPC. Since we desire to make our description as complete as possible for a scientific audience, cf. Section IV-A, we have gone into perhaps more detail than is needed and so perhaps given the impression that the Newton-Raphson code is more complex than it actually is. In practise the GNU sqrt code can be easily adapted. Secondly, for speed, the code uses a data-driven approach for the Newton-Raphson derivative. Both of these were manual choices for the PowerPC. Future research might investigate using genetic improvement on the code in order to see if this is optimal on other computers, if the data-derivative approach should be used in drcp, or if GI can find other optimisations.

Although software maintenance may require highly skilled experts [46, page 65], it may be that the maintenance is not difficult, but the human resources available are already stretched thin. In either case AI may be useful.

C. Future work: GI Autoport Test Cases and Test Oracles

The GNU C library includes more than 6000 extensive test suites. These include the square root test cases, of 599 individual tests. Like other glibc `math` functions, they are used for complex numbers (i.e. with real and imaginary parts), different precisions (long double, double, float etc.) and inlined and noninlined code. Just concentrating upon testing sqrt() (i.e. `double`), the tests are executed 1360 times (plus 1348 tests for errors and exceptions). Although AI has made considerable progress in generating test cases to exercise code [47], [48] usually this relies on implicit oracles [49]–[53] (such as: does the test cause the code to crash with a null pointer exception?), or approximate oracles such as: does the output include strings such as “error”, “problem” or “exception” [54, page 262]? We have used traditional manual testing to demonstrate our drcp, however, what if we were to use not just the existing implementation but the existing test suite and use it to test the new functionality? How much adaptation of the tests would be need? How much of this could be automated? Would genetic improvement be able to evolve existing test cases and their test *oracles*? The little work on automatic test case porting [55] and the presence of thousands of test suites with hundreds tests and their test oracles makes using GI to transplant glibc test suites a tempting target for future research.

VI. Conclusions

The cost of software maintenance is staggering. Although support tools are common place, it remains an essentially tedious error prone manual process with little existing evolutionary computing (EC) research. Indeed although recognised since the early 1980's there is little research on automatic ways to maintain numeric values even though this is an important part of software maintenance.

Most EC software engineering research has concentrated upon source code. Although we already have a few examples of GI programs in use and under regular software maintenance [56] [57] [11] [6], [58], there is a fear that some in the IT industry might be resistant to AI automated source code improvement. Since software developers care about their source code, potentially, by concentrating automatic updates on parameters within code, rather than the instructions, it may be that they will be more accepting of evolved artefacts.

Previously we showed one example where evolutionary computation was used to improve the accuracy of an existing program by automatically maintaining numeric values within it. More recently we showed an example where EC was used to transplant data to give new functionality. We claimed at the time that the approach was more general and here we have further demonstrated it to give a double precision implementation of division. Although primarily a further demonstration of the power of the approach, in some cases, particularly for internet-of-things mote low resource computing, the evolved implementation could be competitive.

Acknowledgements

I am grateful for the assistance of Justyna Petke and Roy Longbottom.

References

[1] S. Hanson *et al.*, "A low-voltage processor for sensing applications with picowatt standby mode," *IEEE Journal of Solid-State Circuits*, vol. 44, no. 4, pp. 1145–1155, April 2009. http://dx.doi.org/10.1109/JSSC.2009.2014205

[2] *ARM1176JZF-S Technical Reference Manual*, Revision: r0p7 ed., 2009. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0301h/DDI0301H_arm1176jzfs_r0p7_trm.pdf

[3] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, no. 8, pp. 114–117, April 19 1965.

[4] M. Harman and B. F. Jones, "Search based software engineering," *Information and Software Technology*, vol. 43, no. 14, pp. 833–839, Dec. 2001. http://dx.doi.org/10.1016/S0950-5849(01)00189-6

[5] M. Harman *et al.*, "The GISMOE challenge: Constructing the Pareto program surface using genetic programming to find better programs," in *The 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 12)*. Essen, Germany: ACM, Sep. 3-7 2012, pp. 1–14. http://dx.doi.org/10.1145/2351676.2351678

[6] A. Marginean *et al.*, "SapFix: Automated end-to-end repair at scale," in *41st International Conference on Software Engineering*, J. M. Atlee and T. Bultan, Eds. Montreal: ACM, 25-31 May 2019, pp. 269–278. http://dx.doi.org/10.1109/ICSE-SEIP.2019.00039

[7] J. R. Koza, *Genetic Programming: On the Programming of Computers by Natural Selection*. MIT press, 1992.

[8] J. R. Koza *et al.*, "What's AI done for me lately? genetic programming's human-competitive results," *IEEE Intelligent Systems*, vol. 18, no. 3, pp. 25–31, May/Jun. 2003. http://dx.doi.org/10.1109/MIS.2003.1200724

[9] C. Le Goues *et al.*, "The case for software evolution," in *Proceedings of the FSE/SDP workshop on Future of software engineering research, FoSER'10*, G.-C. Roman and K. J. Sullivan, Eds. Santa Fe, New Mexico, USA: ACM, Nov. 7-11 2010, pp. 205–210. http://dx.doi.org/10.1145/1882362.1882406

[10] ——, "Automated program repair," *Communications of the ACM*, vol. 62, no. 12, pp. 56–65, Dec. 2019. http://dx.doi.org/10.1145/3318162

[11] S. O. Haraldsson *et al.*, "Fixing bugs in your sleep: How genetic improvement became an overnight success," in *GI-2017*, J. Petke *et al.*, Eds. Berlin: ACM, 15-19 Jul. 2017, pp. 1513–1520, best paper. http://dx.doi.org/10.1145/3067695.3082517

[12] D. R. White *et al.*, "Evolutionary improvement of programs," *IEEE Transactions on Evolutionary Computation*, vol. 15, no. 4, pp. 515–538, Aug. 2011. http://dx.doi.org/10.1109/TEVC.2010.2083669

[13] W. B. Langdon and M. Harman, "Optimising existing software with genetic programming," *IEEE Transactions on Evolutionary Computation*, vol. 19, no. 1, pp. 118–135, Feb. 2015. http://dx.doi.org/10.1109/TEVC.2013.2281544

[14] J. Petke *et al.*, "Genetic improvement of software: a comprehensive survey," *IEEE Transactions on Evolutionary Computation*, vol. 22, no. 3, pp. 415–432, Jun. 2018. http://dx.doi.org/doi:10.1109/TEVC.2017.2693219

[15] D. R. White *et al.*, "Deep parameter tuning of concurrent divide and conquer algorithms in Akka," in *20th European Conference on the Applications of Evolutionary Computation*, ser. Lecture Notes in Computer Science, G. Squillero and Kevin Sim, Eds., vol. 10200. Amsterdam: Springer, 19-21 Apr. 2017, pp. 35–48. http://dx.doi.org/10.1007/978-3-319-55792-2_3

[16] E. Schulte *et al.*, "Post-compiler software optimization for reducing energy," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'14*. Salt Lake City, Utah, USA: ACM, 1-5 Mar. 2014, pp. 639–652. http://dx.doi.org/10.1145/2541940.2541980

[17] B. R. Bruce, "Energy optimisation via genetic improvement a SBSE technique for a new era in software development," in *Genetic Improvement 2015 Workshop*, W. B. Langdon *et al.*, Eds. Madrid: ACM, 11-15 Jul. 2015, pp. 819–820. http://dx.doi.org/10.1145/2739482.2768420

[18] N. Burles *et al.*, "Object-oriented genetic improvement for improved energy consumption in Google Guava," in *SSBSE*, ser. LNCS, Y. Labiche and M. Barros, Eds., vol. 9275. Bergamo, Italy: Springer, Sep. 5-7 2015, pp. 255–261. http://dx.doi.org/10.1007/978-3-319-22183-0_20

[19] Fan Wu *et al.*, "Deep parameter optimisation," in *GECCO '15: Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, S. Silva *et al.*, Eds. Madrid: ACM, 11-15 Jul. 2015, pp. 1375–1382. http://dx.doi.org/10.1145/2739480.2754648

[20] A. Marginean *et al.*, "Automated transplantation of call graph and layout features into Kate," in *SSBSE*, ser. LNCS, Y. Labiche and M. Barros, Eds., vol. 9275. Bergamo, Italy: Springer, Sep. 5-7 2015, pp. 262–268. http://dx.doi.org/10.1007/978-3-319-22183-0_21

[21] E. T. Barr *et al.*, "Automated software transplantation," in *International Symposium on Software Testing and Analysis, ISSTA 2015*, Tao Xie and M. Young, Eds. Baltimore, Maryland, USA: ACM, 14-17 Jul. 2015, pp. 257–269, ACM SIGSOFT Distinguished Paper Award. http://dx.doi.org/10.1145/2771783.2771796

[22] M. Harman *et al.*, "Babel Pidgin: SBSE can grow and graft entirely new functionality into a real world system," in *Proceedings of the 6th International Symposium, on Search-Based Software Engineering, SSBSE 2014*, ser. LNCS, C. Le Goues and Shin Yoo, Eds., vol. 8636. Fortaleza, Brazil: Springer, 26-29 Aug. 2014, pp. 247–252, winner SSBSE 2014 Challange Track. http://dx.doi.org/10.1007/978-3-319-09940-8_20

[23] Yue Jia *et al.*, "Grow and serve: Growing Django citation services using SBSE," in *SSBSE 2015 Challenge Track*, ser. LNCS, Shin Yoo and L. Minku, Eds., vol. 9275. Bergamo, Italy: Springer, 5-7 Sep. 2015, pp. 269–275. http://dx.doi.org/10.1007/978-3-319-22183-0_22

[24] W. B. Langdon and M. Harman, "Grow and graft a better CUDA pknotsRG for RNA pseudoknot free energy calculation," in *Genetic Improvement 2015 Workshop*, W. B. Langdon *et al.*, Eds. Madrid: ACM, 11-15 Jul. 2015, pp. 805–810. http://dx.doi.org/10.1145/2739482.2768418

[25] W. B. Langdon, "Genetic improvement of software for multiple objectives," in *SSBSE*, ser. LNCS, Y. Labiche and M. Barros, Eds., vol.

9275. Bergamo, Italy: Springer, Sep. 5-7 2015, pp. 12–28, invited keynote. http://dx.doi.org/10.1007/978-3-319-22183-0_2

[26] W. B. Langdon and M. Harman, "Evolving a CUDA kernel from an nVidia template," in *2010 IEEE World Congress on Computational Intelligence*, P. Sobrevilla, Ed. Barcelona: IEEE, 18-23 Jul. 2010, pp. 2376–2383. http://dx.doi.org/10.1109/CEC.2010.5585922

[27] A. E. I. Brownlee *et al.*, "Gin: genetic improvement research made easy," in *GECCO '19: Proceedings of the Genetic and Evolutionary Computation Conference*, M. Lopez-Ibanez *et al.*, Eds. Prague, Czech Republic: ACM, 13-17 Jul. 2019, pp. 985–993. http://dx.doi.org/10.1145/3321707.3321841

[28] Gabin An *et al.*, "PyGGI 2.0: Language independent genetic improvement framework," in *Proceedings of the 27th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering ESEC/FSE 2019)*, S. Apel and A. Russo, Eds. Tallinn, Estonia: ACM, Aug. 26–30 2019, pp. 1100–1104. http://dx.doi.org/10.1145/3338906.3341184

[29] W. B. Langdon, "Genetic improvement GISMOE blue software tool demo," University College, London, London, UK, Tech. Rep. RN/18/06, 22 Sep. 2018.
http://www.cs.ucl.ac.uk/fileadmin/user_upload/blue.pdf

[30] M. Orlov and M. Sipper, "Flight of the FINCH through the Java wilderness," *IEEE Transactions on Evolutionary Computation*, vol. 15, no. 2, pp. 166–182, Apr. 2011. http://dx.doi.org/10.1109/TEVC.2010.2052622

[31] E. Schulte *et al.*, "Automated program repair through the evolution of assembly code," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*. Antwerp: ACM, 20-24 Sep. 2010, pp. 313–316. http://dx.doi.org/10.1145/1858996.1859059

[32] ——, "Repairing COTS router firmware without access to source code or test suites: A case study in evolutionary software repair," in *Genetic Improvement 2015 Workshop*, W. B. Langdon *et al.*, Eds. Madrid: ACM, 11-15 Jul. 2015, pp. 847–854, best Paper. http://dx.doi.org/10.1145/2739482.2768427

[33] W. B. Langdon and J. Petke, "Software is not fragile," in *Complex Systems Digital Campus E-conference, CS-DC'15*, ser. Proceedings in Complexity, P. Parrend *et al.*, Eds. Springer, Sep. 30-Oct. 1 2015, pp. 203–211, invited talk. http://dx.doi.org/10.1007/978-3-319-45901-1_24

[34] R. J. Martin and W. M. Osborne, "Guidance on software maintenance," National Bureau of Standards, Department of Commerce, Washington DC, USA, NBS Special Publication 500-106, Dec 1983. http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nbsspecialpublication500-106.pdf

[35] M. Mohan and D. Greer, "A survey of search-based refactoring for software maintenance," *Journal of Software Engineering Research and Development*, vol. 6, no. 3, 7 February 2018. http://dx.doi.org/10.1186/s40411-018-0046-4

[36] F. G. de Freitas and J. T. de Souza, "Ten years of search based software engineering: A bibliometric analysis," in *Third International Symposium on Search based Software Engineering (SSBSE 2011)*, ser. LNCS, M. B. Cohen and M. O Cinneide, Eds., vol. 6956. Szeged, Hungary: Springer, 10th - 12th September 2011, pp. 18–32. http://dx.doi.org/10.1007/978-3-642-23716-4_5

[37] R. Lorenz, S. H. Bernhart, C. Höner zu Siederdissen, H. Tafer, C. Flamm, P. F. Stadler, and I. L. Hofacker, "ViennaRNA package 2.0," *Algorithms for Molecular Biology*, vol. 6, no. 1, 2011. http://dx.doi.org/10.1186/1748-7188-6-26

[38] W. B. Langdon *et al.*, "Evolving better RNAfold structure prediction," in *EuroGP 2018: Proceedings of the 21st European Conference on Genetic Programming*, ser. LNCS, M. Castelli *et al.*, Eds., vol. 10781. Parma, Italy: Springer Verlag, 4-6 Apr. 2018, pp. 220–236. http://dx.doi.org/10.1007/978-3-319-77553-1_14

[39] W. B. Langdon and J. Petke, "Evolving better software parameters," in *SSBSE 2018 Hot off the Press Track*, ser. LNCS, T. E. Colanzi and P. McMinn, Eds., vol. 11036. Montpellier, France: Springer, 8-9 Sep. 2018, pp. 363–369. http://dx.doi.org/10.1007/978-3-319-99241-9_22

[40] ——, "Genetic improvement of data gives binary logarithm from sqrt," in *GECCO '19: Proceedings of the Genetic and Evolutionary Computation Conference Companion*, R. Allmendinger *et al.*, Eds. Prague, Czech Republic: ACM, 13-17 Jul. 2019, pp. 413–414. http://dx.doi.org/10.1145/3319619.3321954

[41] W. B. Langdon, "Genetic improvement of data gives double precision invsqrt," in *7th edition of GI @ GECCO 2019*, B. Alexander *et al.*, Eds. Prague, Czech Republic: ACM, Jul. 13-17 2019, pp. 1709–1714. http://dx.doi.org/10.1145/3319619.3326800

[42] P. W. Markstein, "Computation of elementary functions on the IBM RISC System/6000 processor," *IBM Journal of Research and Development*, vol. 34, no. 1, pp. 111–119, Jan 1990. http://dx.doi.org/10.1147/rd.341.0111

[43] N. Hansen and A. Ostermeier, "Completely derandomized self-adaptation in evolution strategies," *Evolutionary Computation*, vol. 9, no. 2, pp. 159–195, Summer 2001. http://dx.doi.org/10.1162/106365601750190398

[44] O. Krauss and W. B. Langdon, "Automatically evolving lookup tables for function approximation," in *EuroGP 2020: Proceedings of the 23rd European Conference on Genetic Programming*, Ting Hu *et al.*, Eds. Seville, Spain: Springer Verlag, 15-17 Apr. 2020, forthcoming.

[45] E. W. Dijkstra, *"Testing shows the presence, not the absence of bugs." in Software Engineering Techniques: Report of a conference sponsored by the NATO Science Committee*, Robert M. McClure, 2001 ed. Rome, Italy: NATO, Scientific Affairs Division, Brussels, 27-31 Oct 1969, ch. 3.1, p. 16. http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1969.PDF

[46] S. M. H. Dehaghani and N. Hajrahimi, "Which factors affect software projects maintenance cost more?" *Acta Informatica Medica*, vol. 21, no. 1, pp. 63–66, Mar 2013. http://dx.doi.org/10.5455/AIM.2012.21.63-66

[47] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in 8^{th} *European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE '11)*. Szeged, Hungary: ACM, September 5th - 9th 2011, pp. 416–419. http://dx.doi.org/10.1145/2025113.2025179

[48] N. Alshahwan *et al.*, "Deploying search based software engineering with Sapienz at Facebook," in *SSBSE 2018*, ser. LNCS, T. E. Colanzi and P. McMinn, Eds., vol. 11036. Montpellier, France: Springer, 8-9 Sep. 2018, pp. 3–45. http://dx.doi.org/10.1007/978-3-319-99241-9_1

[49] E. T. Barr *et al.*, "The oracle problem in software testing: A survey," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507–525, May 2015. http://dx.doi.org/109/TSE.2014.2372785

[50] M. Pezze and Cheng Zhang, "Automated test oracles: A survey," in *Advances in Computers*. Elsevier, 2015, vol. 95, pp. 1–48. http://dx.doi.org/10.1016/B978-0-12-800160-8.00001-2

[51] P. A. Nardi and E. F. Damasceno, "A survey on test oracles," *Journal on Advances in Theoretical and Applied Informatics*, vol. 1, no. 2, pp. 50–59, 2015. http://revista.univem.edu.br/index.php/jadi/article/view/1034

[52] G. Jahangirova *et al.*, "Test oracle assessment and improvement," in *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA'16)*. Saarbruecken, Germany: ACM, 2016, pp. 247–258. http://dx.doi.org/10.1145/2931037.2931062

[53] W. B. Langdon *et al.*, "Inferring automatic test oracles," in *Search-Based Software Testing*, J. P. Galeotti and J. Petke, Eds., Buenos Aires, Argentina, 22-23 May 2017, pp. 5–6. http://dx.doi.org/10.1109/SBST.2017.1

[54] A. I. Esparcia-Alcazar *et al.*, "Using genetic programming to evolve action selection rules in traversal-based automated software testing: results obtained with the TESTAR tool," *Memetic Computing*, vol. 10, no. 3, pp. 257–265, Sep. 2018. http://dx.doi.org/10.1007/s12293-018-0263-8

[55] Tianyi Zhang and Miryung Kim, "Automated transplantation and differential testing for clones," in *Proceedings of the 39th International Conference on Software Engineering*. Buenos Aires, Argentina: IEEE Press, 2017, pp. 665–676. http://dx.doi.org/10.1109/ICSE.2017.67

[56] W. B. Langdon and Brian Yee Hong Lam, "Genetically improved BarraCUDA," *BioData Mining*, vol. 20, no. 28, 2 Aug. 2017. http://dx.doi.org/10.1186/s13040-017-0149-1

[57] W. B. Langdon and R. Lorenz, "Improving SSE parallel code with grow and graft genetic programming," in *GI-2017*, J. Petke *et al.*, Eds. Berlin: ACM, 15-19 Jul. 2017, pp. 1537–1538. http://dx.doi.org/10.1145/3067695.3082524

[58] N. Alshahwan, "Industrial experience of genetic improvement in Facebook," in *GI-2019, ICSE workshops proceedings*, J. Petke *et al.*, Eds. Montreal: IEEE, 28 May 2019, p. 1, invited Keynote. http://dx.doi.org/10.1109/GI.2019.00010