

Genetic Improvement of Data for Maths Functions

WILLIAM B. LANGDON, CREST, Department of Computer Science, University College London, UK
OLIVER KRAUSS, University of Applied Sciences Upper Austria, Austria

We use continuous optimisation and manual code changes to evolve up to 1024 Newton-Raphson numerical values embedded in an open source GNU C library glibc square root sqrt to implement a double precision cube root routine cbrt, binary logarithm log2 and reciprocal square root function for C in seconds. The GI inverted square root $x^{-\frac{1}{2}}$ is far more accurate than Quake's InvSqrt, Quare root. GI shows potential for automatically creating mobile or low resource mote smart dust bespoke custom mathematical libraries with new functionality.

CCS Concepts: • **Software and its engineering** → **Search-based software engineering**.

Additional Key Words and Phrases: evolutionary computing, software engineering, search based software engineering, SBSE, software maintenance of empirical constants, data transplantation, glibc, vector normalisation, Newton's method

ACM Reference Format:

William B. Langdon and Oliver Krauss. 2021. Genetic Improvement of Data for Maths Functions. *ACM Transactions on Evolutionary Learning and Optimization*, 1, 2, Article 7 (July 2021), 30 pages. <https://doi.org/10.1145/3461016>

1 NEW FUNCTIONALITY VIA DATA UPDATE

Even more than forty years after the advent of software engineering, we are still faced with an industry reliant on human labour. Most research (including genetic improvement GI [61], [31], [50]) concentrates on the code itself. There has been relatively little effort in automating other aspects of software development such as parameters embedded in the software. We wish to find a radically new automatic approach to maintaining the numeric and data components of software, which will yield an increase in human productivity, giving rise to both cost savings and also significantly reducing delays in introducing new system components.

We provide a small number of examples of converting existing mathematical functions into new ones. These can be viewed mostly as a demonstration of what evolution can do. We hope that they will encourage future research.

2 MAINTAINING NUMBERS WITHIN CODE

Many programs contain embedded parameters. Typically these are numeric values, often float or double, but also integers, e.g. the GNU C library contains more than a million integer constants (see Figure 1, also [34]). In many cases these parameters relate to the software itself or to simple facts which are unlikely to change during the program's lifetime or period of active use. However, many others ought to be updated. This maintenance problem has been known for a long time

Authors' addresses: William B. Langdon, W.Langdon@cs.ucl.ac.uk, CREST, Department of Computer Science, University College London, Gower Street, London, UK, WC1E 6BT; Oliver Krauss, University of Applied Sciences Upper Austria, Softwarepark 11, Hagenberg, Austria, 4232, oliver.krauss@fh-hagenberg.at.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.

2688-3007/2021/7-ART7 \$Preprint

<https://doi.org/10.1145/3461016>

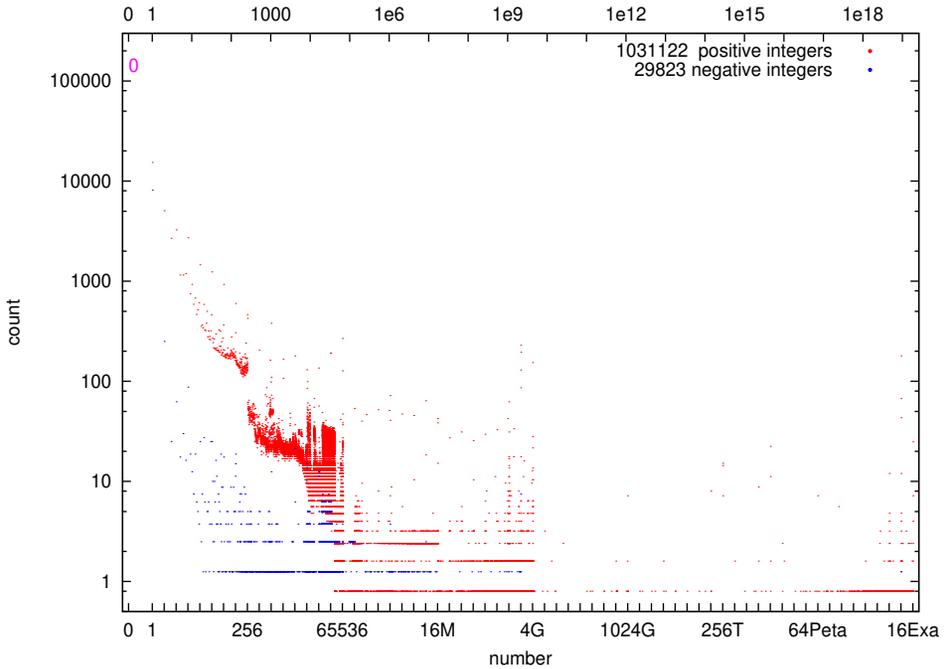


Fig. 1. The GNU C library, excluding test suite, contains 1 202 711 integer constants. Zero is the most common, occurring a total of 141,874 times, followed by 1 (19 203) and -1 (6 479). Every integer between -28 and 40 956 occurs at least once. To avoid overlap, positive and negative values are slightly offset vertically.

(Martin and Osborne, 1983 [45, Section 6.8, page 24, Hard Coded Parameters Which Are Subject To Change]).

Parameters may relate to heuristics within the code, which the developer chose before contact with real users. Their values perhaps should have been updated shortly after first release. Also values (e.g. those relating to memory or array sizes) may need updating due to operating on new hardware, as well as to changes in patterns of use. Other parameters can relate to the problem itself. For example, chemical reaction rate constants in ozone layer simulations [11]. In some cases the exact numerical values are critical [11]. Some physical values are known with very high precision, but for others the state of scientific knowledge can improve over the operational life of the program. For example, the ViennaRNA package [39] contains more than 50 000 energy binding values. These are derived from scientific measurements of RNA molecules. Even so, during the relatively short life of this suite of C programs, knowledge has moved on and various newer versions of these parameters are available. Recently [38] we showed genetic improvement (GI) could be used to adapt these 50 000 `int` values. (The GI values have been distributed with ViennaRNA since version 2.4.5.) In [38] we used custom mutation and crossover operators to evolve compile time constants within the C source code. Evolution took about five days (rather than a few minutes or seconds in the following examples, Sections 5, 6 and 7) to find a new program which on thousands of real examples gave predictions which were on average 11% more accurate. (Although some were worse, most were unchanged or better.) Notice that there were no changes to the code. Only data were changed.

As computing is now mature, maintaining software has become the dominant cost. (Marounek [44, page 51] quotes figures of more than 90% of total cost.) Moreover, software maintenance routinely requires highly skilled experts [15, page 65]. Yet a recent survey [46] starts by saying “a relatively

small amount [of search based software engineering (SBSE) research] is related to software maintenance”, whilst de Freitas and de Souza [14] do not give a breakdown of the SBSE literature on software maintenance. Indeed it appears that maintaining embedded constants within existing packages has received little attention so far. For example, Butler, e.g. [10], considers the maintenance impact of names given to constants in Java source code, but not how to maintain their values. Similarly, Tiella and Ceccato [58] consider how to hide constant values, but not how to update them.

There is some research on parameter tuning, e.g. the ParamILS¹ and irace² tools. However, there is scarcely any on updating parameters in the code that are not specifically exposed to the user for tuning. The deep parameter tuning work by Wu et al. [67] being the first known example, where they optimised for runtime and memory consumption. Unlike Wu et al. [67], we focus on adapting numerical values. Previous work on evolving new features using GI, either transplanted portions of one program to another, Marginean et al. [4, 42], or used our grow-and-graft approach [18, 20, 25, 30]. Marginean et al.’s highly innovative source code transplantation allows automatic transfer of source code. For example, additional features not currently in the popular open source C++ editor Kate (such as source code indentation and C call graph layout) were taken from existing open source code, and using TXL, automatically grafted into Kate. Genetic programming was used to automatically match variables in the host and donor source code. Grow-and-graft evolves the new functionality separately (rather than taking it from human written open source code) and then adds it to existing code (as with automated software transplantation). For example, Pidgin is a large mature open source instant messaging system written in C. Externally we evolved (“grew”) a module which, using Google Translate, translated English text into Portuguese and Korean. This was automatically “grafted” into Pidgin. Our approach here does not require additional code, just changes within the existing code base. Although here code changes are required, we seek to encourage research into automated update of embedded data with a few examples.

3 GI CREATING NEW FUNCTIONALITY WITH DATA CHANGES

Most Genetic Improvement research [23, 26, 48–50] has applied evolution to program code. Usually source code, but also byte code [47], assembler [53] and even machine code [54] have been optimised. A suitable fitness function is essential for such directed evolution. The fitness function may consider functionality (as here) and/or non-functional properties, such as run-time, energy [6, 9, 52, 62] or even memory [67] consumption. In [37] we considered the problems of using physical measurements as part of the fitness measure for GI (see also [55]). We apply search based techniques [19] directly to data values embedded inside the source code, with a view to create new functionality rather than to improve existing functionality (see Table 1).

In the next section (4) we describe how an existing open source C function within the GNU C mathematics library implements double precision sqrt using Newton-Raphson. Section 5, cbrt, Section 6, log2, and Section 7, invsqrt, present evolving 512 or 1024 constants to give mathematical functions. Sections 5.3, 6.4 and 7.4 find accuracy is typically better than one bit in the IEEE 754 double precision representation, Figure 2, and is never worse than double precision requirements. Additional motivation, discussion of limitations and possible extensions in Section 8 are followed by conclusions (Section 9), in which we suggest there is a great need for research into both automated data update and data transplantation.

¹<http://www.cs.ubc.ca/labs/beta/Projects/ParamILS/>

²<http://iridia.ulb.ac.be/irace/>

Table 1. Applying Genetic Improvement to Data Embedded in Source Code

<ul style="list-style-type: none"> ● Establish goal <p>For example, do we wish to maintain an existing program so that it gives better answers (a functional change). Or do we need to ensure it retains its behaviour (non-functional change) but is better in some measurable way? E.g., it is quicker. Alternatively do we want to transform it to do something different?</p> ● Locate data to be updated and program test cases. <ul style="list-style-type: none"> – Define optimisation objective measure. <p>The objective fitness function will be driven by the goal. Although subjective user driven improvement, via interactive evolution [57] could be considered, mostly we refer to automatic optimisation, which requires one or more metrics to drive the search.</p> <p>Remember that the point of the fitness function is to guide search, not to prove that a trial solution is correct. As the fitness function is used many times, it may be better to delay validation until after the search. It may help to subsample the available test data and only use a small random sample, which can be changed during search. Similarly, if run time depends on the program’s input data it may be better to choose multiple input data which give short run times rather than sampling uniformly. That is, it may be better to delay uniform sampling until the post-optimisation validation stage. If test case data are to be used to validate the optimised program, remember, to avoid over fitting, to keep a clear separation from input data used for training and input data used for validation [13].</p> – Define representation. <p>What type of embedded data are you hoping to optimise? Is it homogeneous? Does it have special properties (e.g. multiple of 10, symmetric matrices)?</p> <p>Can you operate directly on the embedded data, or would an intermediate representation help?</p> – Choose search operator(s). <p>If the embedded data are continuous and homogeneous, will Gaussian mutation be sufficient? If a mixture of float and Boolean, will a problem specific search operator be needed? Is there structure in the source code data which can be exploited?</p> <p>Will a crossover operator help by mixing together good parts of different solutions?</p> <p>Where problems are new, it may be worth using multiple different mutation and crossover operators. It may be that the loss in search efficiency from not using the “correct” “optimal” search operator(s) is small.</p> – Choose optimisation tool. <p>Obviously the above choices interact. It may be sensible to choose a representation, a tool, etc. that you are familiar with and trust rather than starting by seeking optimal choices.</p> ● Apply optimisation tool ● Evaluate results. <p>If the results are not satisfactory, locate the problem and reiterate the corresponding steps above. Otherwise stop.</p>

4 AUTOMATED PARAMETER TUNING FOR EVOLVING NEW FUNCTIONALITY

Although there are analytic approaches, all three double precision mathematical functions (cube root $\sqrt[3]{x}$, binary logarithm \log_2 , Section 6, and and reciprocal square root $x^{-1/2}$, Section 7) are based on an existing open source C implementation of the square root function and then using minimal

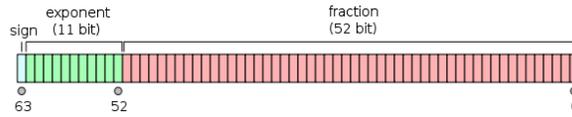


Fig. 2. IEEE 754 Double-precision floating-point format (Wikipedia). Notice sign (bit 63, light blue) is zero for positive numbers.

manual code changes plus artificial evolution (in the form of CMA-ES [17]) to mutate 512 or 1024 constant values for square root into those needed for the new function.

The GNU C library (<https://www.gnu.org/s/libc/>, 1 250 944 lines of code) contains many implementations of the square root function (`sqrt`). Most simply call the underlying hardware's square root function. However some versions of the PowerPC do not have `sqrt` implemented in hardware. For these the GNU library (`sysdeps/powerpc/fpu`) provides a C implementation based on table lookup and Newton-Raphson approximation [43]. The GNU implementation exploits the format of double precision numbers and Newton-Raphson's rapid convergence to give an efficient double precision implementation of `sqrt` which does not need special hardware. This version of `sqrt()` was selected for use as the start for table-based C implementations of the cube root `cbrt()` $\sqrt[3]{x}$, `log2()` and `invsqrt()` $x^{-1/2}$ double precision functions.

4.1 Newton-Raphson

Newton-Raphson, invented by Sir Isaac Newton and refined by Joseph Raphson, is an iterative approximation method for finding the roots of continuous differentiable functions. Essentially we start with an initial guess x_1 for where the root is (see Figure 3). (These start points are hard coded float constants in `glibc`'s table-based `sqrt`.) At x_1 we calculate the value of the function $f(x_1)$. If we were spot on with our initial guess $f(x_1) = 0$ and so the root we seek is x_1 and we can stop. Typically we test $f(x_1)$ and if it is sufficiently close to zero we stop. If not, we use the value of $f(x_1)$ and its derivative $f'(x_1)$ (sloping red line in Figure 3) to make a new estimate x_2 of where the root is, $x_2 = x_1 - \frac{f(x_1)}{f'(x_1)}$. Again we test $f(x_2)$ to see if x_2 is sufficiently close and we should stop. In general $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$ and we test each new $f(x_n)$ until we get a sufficiently accurate answer.

4.2 GNU C Library `sqrt`'s use of Newton-Raphson

The table driven PowerPC version of `sqrt`, `e_sqrt.c`, Appendix A, uses the format of double precision numbers (Figure 4). Since $\sqrt{2^k} = 2^{k/2}$, to find the square root of the exponent part of x , `e_sqrt.c` uses a shift right operation to divide the exponent by two. Secondly it treats the least significant bit of the exponent plus the fractional part as the normalised version of x lying between 0.5 and 2.0.

`e_sqrt.c` uses the top nine bits of the normalised number as an index into a table of 512 pairs of floating point numbers. The first of each pair is used as a start point for the Newton-Raphson method to find a root of $f(x) = x^2 - a$ (where a is the input to `sqrt`, i.e. we seek $x = a^{1/2}$).

As described above in Section 4.1, Newton's method requires repeated division by the derivative. The derivative of $f(x)$ is $f'(x) = 2x$. For speed `e_sqrt.c` uses the second of each pair of entries in the table to store the first estimate of the reciprocal of the derivative, i.e. $1/2x$. By using the reciprocal, the division can be replaced by a (faster) multiplication. Newton-Raphson is also applied to the estimate of the reciprocal of the derivative. `e_sqrt.c` says its `sqrt` "consists of two interleaved Newton-Raphson approximations, one to find the actual square root, and one to find its reciprocal without the expense of a division operation. The tricky bit here is the use of the PowerPC multiply-add operation to get the required accuracy with high speed." Appendix A.

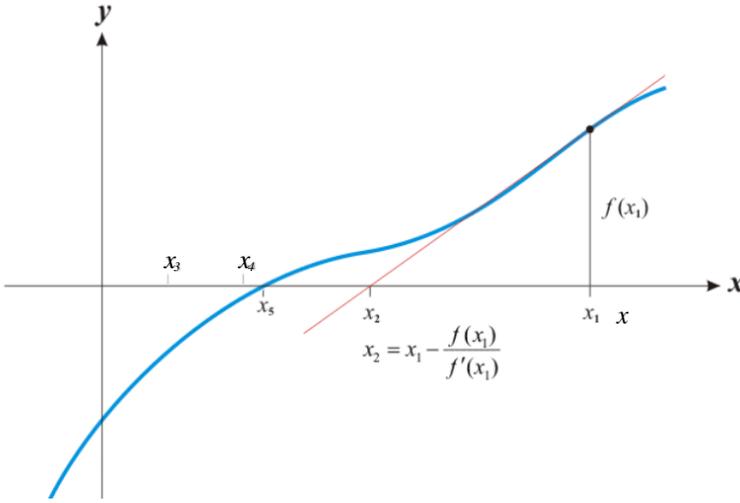


Fig. 3. First iteration of Newton-Raphson (Wikipedia). $f'(x_1)$ is the derivative of f at x_1 (i.e. the tangent, thin red line). Following the tangent at x_1 to where it crosses the horizontal line $y = 0$, gives x_2 . $x_2 = x_1 - \frac{f(x_1)}{f'(x_1)}$. In this example x_2 is closer than x_1 to the root (where thick blue line representing $f(x)$ crosses the line $y = 0$). Next iteration we start at x_2 and calculate $x_3 = x_2 - \frac{f(x_2)}{f'(x_2)}$. In this example x_3 overshoots but x_4 is close and x_5 is almost exact. For reasonable functions Newton-Raphson converges to the root quadratically fast.

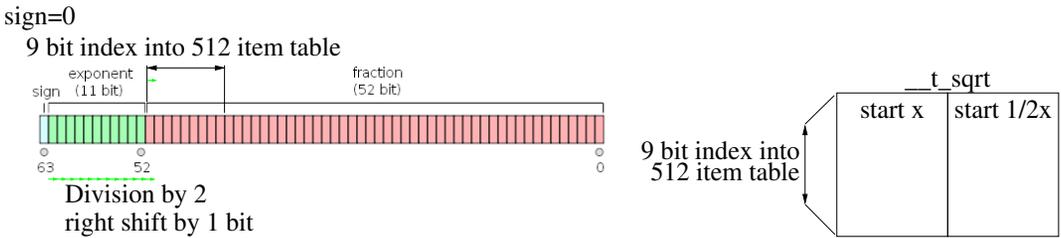


Fig. 4. Left: `e_sqrt.c`, Appendix A, uses right shift of exponent of positive double (bit 63 = 0) to a) divide exponent by two and b) merge least significant bit with top 8 bits of fractional part to give nine bit index. Right: index used with `float t_sqrt` containing 512 x_1 and $1/2x_1$ pairs of initial values for Newton-Raphson iterative solution of \sqrt{x} .

In the PowerPC GNU C library for `sqrt` the next step of the Newton-Raphson iteration for x (i.e. x_{n+1}) is calculated from the current error and the variable `sy`, which holds the reciprocal of the derivative ($1/f'$). See Figure 3 and formulae below:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} = x_n - \frac{x_n^2 - a}{2x_n} = x_n + \frac{1}{2x_n} \times (-\text{error}) = x_n + \text{sy} \times (-\text{error})$$

`e_sqrt.c` double variable $\text{sy} = \frac{1}{2x_n}$

$$\text{error} = f(x_n) = x_n^2 - a$$

Newton-Raphson converges quadratically fast in ideal circumstances. By starting with an 8 or 9 bit approximation each iteration improves the accuracy: 16, 32, and finally 64 bits. Thus for double precision (52 bits, Figure 2) only three iterations are needed. Also for speed `e_sqrt.c` does not check

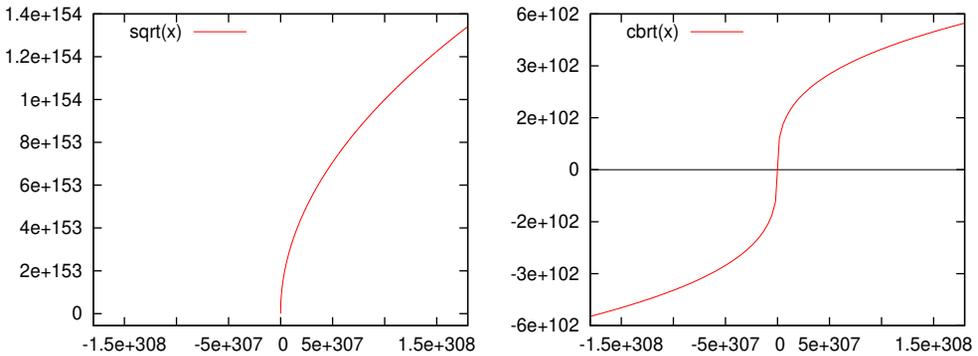


Fig. 5. Left: Double precision square root Right: Double precision cube root

if it has reached the right answer at each iteration but proceeds to do all three iterations in an unrolled loop. The final step is to restore the adjusted exponent $2^{\lfloor k/2 \rfloor}$.

The open source `sqrt` code uses a fast fixed three step Newton-Raphson approach. In addition to an estimate of the derivative of the square root function, Newton-Raphson requires an objective measure, x^2 , to tell it how far it has overshot. Notice, although `e_sqrt.c` uses a table of values, it is not interpolating. Instead it is actually calculating the true square root to double precision accuracy in a small number of steps. For speed, it uses the table of starting values to limit the number of Newton-Raphson iterations needed before it converges on the right answer.

5 OPTIMISING DATA TO GENERATE CUBE ROOT CBRT

Figure 5 shows the `sqrt()` function we start with alongside the cube root function which we evolve (note different vertical scales). Following Table 1, our goal is to create a new function, $\sqrt[3]{x}$, using the already identified data table `__t_sqrt`. The data are readily separated into 512 pairs, so each pair can be optimised separately, and with no need for an indirect representation. The test data and fitness function will be described in Section 5.2.2. We will use CMA-ES (Section 5.2) and its default genetic operators.

5.1 `cbrt` Manual Changes

In the spirit of complete documentation and reproducibility, we list all the manual changes, even though there is overlap between the three new functions.

Whilst in [38] no code changes were needed and here we are primarily concerned with adjusting data values, nevertheless a few changes to the existing PowerPC `sqrt` code (Appendix A) were made by hand so that it could support `cbrt`. (Similar changes were needed for binary logarithm \log_2 , Section 6.2, and and reciprocal square root $x^{-1/2}$, Section 7.2.) For `cbrt`:

- Various powerPC optimisations were disabled.
- Replaced the trap for negative numbers by returning $-\sqrt[3]{-x}$ if x is negative.
- Division of the exponent part of double precision numbers by three is rather more tricky than division by two. Keeping track of the remainder required the multiplication or division by $\sqrt[3]{2}$ or $\frac{2}{\sqrt[3]{2}}$ (Section 5.3). The existing constants `CBRT2` and `SQR_CBRT2` were used.
- The GNU `sqrt` implementation in `sysdeps/powerpc/fpu/e_sqrt.c` that we start from, uses a right shift to do two operations. Firstly to divide the exponent by two. And secondly to combine the least significant bit of the exponent with the top eight bits of the fractional part,

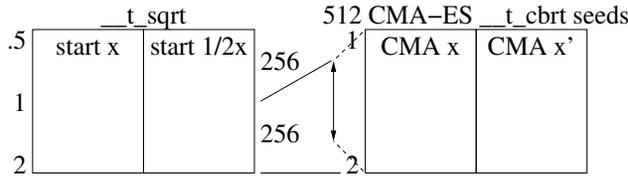


Fig. 6. GI takes half of the open source table for calculating sqrt (`__t_sqrt`, 1..2, left, also Appendix A) and uses it to seed CMA-ES 512 times (`__t_cbrt`, 1..2, right). Since half `__t_sqrt` is 256 pairs of values, GI creates the missing 256 pairs of seeds by interpolating between adjacent pairs.

forming a nine bit index into the table (see Figures 4 and 6), effectively mapping numbers in the range 0.5 to 2 onto the table. The more tricky division by three led to the decision to exclude the exponent and to just use the top nine bits of the fractional part as the table index. So numbers in the range 1 to 2 are mapped onto the table, see also Figure 10.

- The constant `almost_half` was replaced by new constant `almost_third = 0.3333333333333334`.
- As in `e_sqrt.c`, the infrequently used recursive code to deal with tiny denormalised numbers (denorm:) multiplies by 2^{108} . This makes the input bigger by adding 108 to the exponent. To compensate and return the right value, we now divide the result by the cube root of 2^{108} (rather than the square root of 2^{108}). The division is implemented as multiplication by the reciprocal. I.e. multiplication by $\sqrt[3]{2^{-108}} = 2^{-36}$.

5.2 Automatic Changes to `cbrt` Data Table using CMA-ES

The `__t_sqrt` table contains 512 pairs of `float`. The top 256 correspond to numbers in the range 1 to 2. These were used as start points when evolving the 512 pairs of `float` in the new table `__t_cbrt` (see Figure 6).

The Covariance Matrix Adaptation Evolution Strategy algorithm (CMA-ES [17]) was downloaded from <https://github.com/cma-es/c-maes/archive/master.zip> It was set up to fill the new `__t_cbrt` table one pair ($N=2$) at a time. Each pair being initially set to either the corresponding pair of values in `__t_sqrt` or the mean of two adjacent pairs. The initial mutation step sizes used by CMA-ES were both set (pairwise) to 3.0 times the standard deviation calculated from the 512 pairs of numbers in `__t_sqrt`.

5.2.1 `cbrt` CMA-ES Parameters. The CMA-ES defaults (`cmaes_initials.par`) were used, except: the problem size ($N=2$), the initial values and mutation sizes were loaded from `__t_sqrt` (see previous section) and various small values concerned with run termination were set to zero (`stopFitness`, `stopTolFun`, `stopTolFunHist`, `stopTolX`). The initial seed used for pseudo random numbers was also set externally.

5.2.2 `cbrt` Fitness Function. CMA-ES is run separately for each of the 512 bins. Each bin corresponds to a pair of values in `__t_cbrt`. The search for an optimal pair was guided by the fitness function, see Figures 7 and 8. Each time CMA-ES proposed a pair of values, their fitness is assessed by loading them into `__t_cbrt` and running the `cbrt` code with the modified `__t_cbrt` on three test points.

Three test points gives a good tradeoff between simplicity, run time, reliability and avoiding over fitting. They were: the lowest value for the `__t_cbrt` entry, the mid point and the top most value. As we shall see these three points were sufficient to avoid over fitting. The `cbrt` function was called (using the updated `__t_cbrt`) for each and a sub-fitness value calculated with each of the three returned double. The sub-fitnesses were combined by adding them.

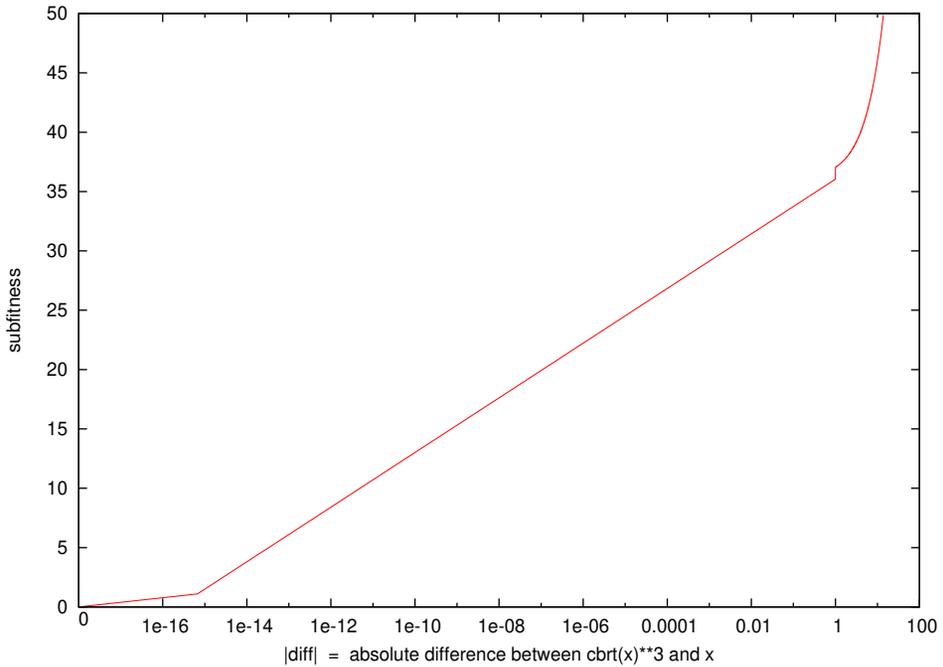


Fig. 7. Plot of one subfitness v. $|\text{diff}|$. Greater than 1 (note discontinuity) subfitness is linear in $|\text{diff}|$ and < 1 it is logarithmic in $|\text{diff}|$. The evolving cbrt is run on each test point x and the difference $|(\text{cbrt}(x))^3 - x|$ found. The goal is to minimise all of the differences. (Note log horizontal scale with perfect solution $|\text{diff}|=0$ in the left hand corner.) Total fitness given by summing subfitness for each of three test points. See also Figure 8.

Each sub-fitness took the output of cbrt , cubed it and took the absolute difference between this and the corresponding test value. If they were the same, the sub-fitness is 0, otherwise it was positive. Since when cbrt is working well, the differences are very small, they were re-scaled for CMA-ES. If the absolute difference was less than one, its log was taken, otherwise the absolute value was used. However, in both cases, to prevent the sub-fitness from being negative, log of the smallest feasible non-zero difference DBL_EPSILON was subtracted. However, more recent work suggests that CMA-ES may not require log scaling [22].

CMA-ES stopped when the difference on all three test points was zero.

5.2.3 cbrt Restart Strategy. When CMA-ES failed to find a pair of values for which all three test cases pass, it was run again with the same initial starting position and mutation size, but a new pseudo random number seed. Mostly CMA-ES found a suitable pair in one run, but in 107 of 512 cases it was run more than once. (In no case was CMA-ES run more than 4 times on a particular pair.)

Figure 9 shows the cbrt optimised values for the pairs of table values compared to the starting seed values taken from the GNU library values (horizontal axis). Although Figure 9 shows, for cbrt , the final values are close to the initial seed values, later work [22] suggests seeding CMA-ES is perhaps not essential and CMA-ES can optimise the table values without seeding.

5.3 Testing the Evolved cbrt Function

The cbrt testing strategy (below) will also be used for binary logarithm \log_2 , Section 6.4, and reciprocal square root $x^{-1/2}$, Section 7.4.

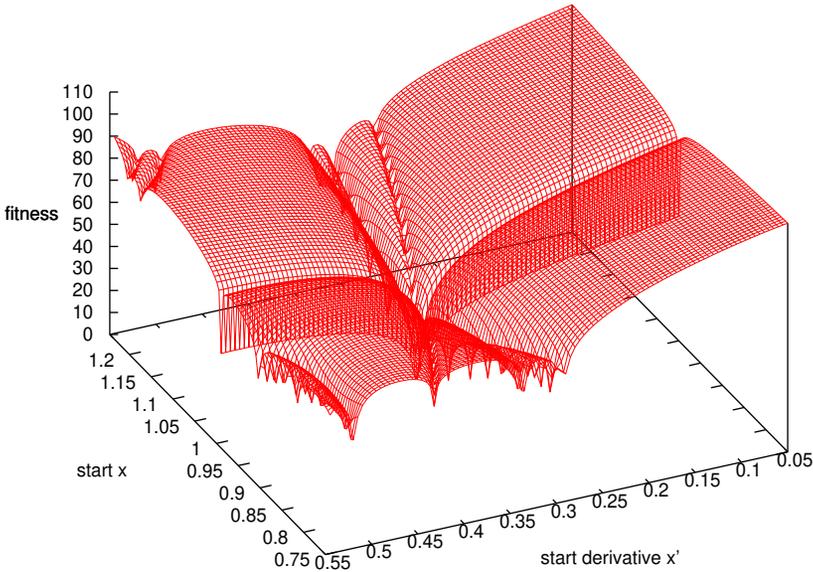


Fig. 8. Example fitness landscape for $\sqrt[3]{x}$ (goal is to minimise). This fitness guides CMA-ES to find a pair of x, x' values for the first bin in `__t_cbrt` ($x=1.0$). The other test points for this bin are $x + 1/1024$, and $\approx x + 1/512$ (i.e. min, mid point and max value). The pair of x, x' values chosen by CMA-ES must give zero error (subfitness = 0, Figure 7) on all three test points for this bin. In total there are 512 bins, which are solved individually.

The pairs of float values found by CMA-ES are shown in Figure 10. The glibc-2.27 powerPC IEEE754 table-based double sqrt function should produce answers within one bit of the correct solution. On 1 536 tests of large integers ($\approx 10^{16}$) designed to test each of the 512 bins 3 times (minimum, maximum and a randomly chosen point) the largest discrepancy between $(\text{cbrt}(x)**3)$ and x was three (i.e. $6.66 \cdot 10^{-16}$). In all tests, including those described in the rest of this section, this only arose when the exponent part of the double was not a multiple of 3. This requires the cbrt code to do an extra multiply or divide by $\sqrt[3]{2}$ or $\sqrt[3]{2}^2$ (i.e. CBRT2 or SQR_CBRT2, see Section 5.1), apparently resulting in additional loss of precision.

As well as ad-hoc testing, and the large positive integer tests mentioned in the previous paragraph, cbrt was tested with 5 120 random numbers uniformly distributed between 1 and 2 (the largest deviation was two³), 5 120 random scientific notation numbers (e.g. 1.998343e-302) and 5 120 random 64 bit patterns. Half the random scientific notation numbers were negative and half positive. 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 cbrt correctly returned NAN. In most cases cbrt returned a double, which when cubed was its input or within one bit of it. In some cases the cubed answer was two from the input. The maximum deviation was three.

6 LOG2

Unlike the deep parameter tuning work by Wu et al. [67], our goal is not to improve existing functionality but to use the framework to create new functionality. Also it is related to data

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

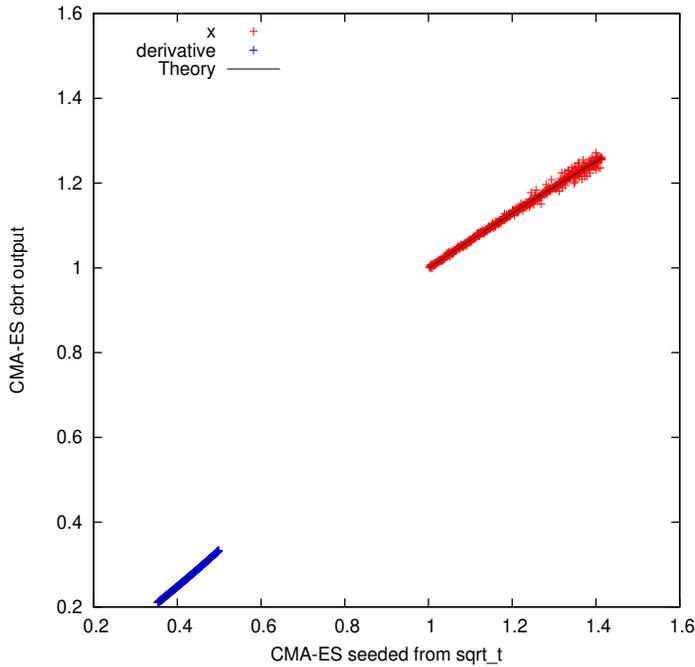


Fig. 9. Plot showing modification from sqrt table values (horizontal axis) to give corresponding cbirt table value (vertical axis). 512 CMA-ES runs. “Theory” lines given by analytic approaches comparing \sqrt{x} and $\sqrt[3]{x}$ and their derivatives.

transplanting as almost all the changes are related to numbers rather than to code (c.f. program transplanting [42],[41]) as data are transferred from the square root function to \log_2 (Figure 11). However they are heavily adapted by CMA-ES [17] once in their new home.

It should be pointed out that \log is a very well known function and there are existing computationally efficient ways to calculate it. Here we use it only as an example.

We extend the interesting class of programs that can be created from existing programs primarily by automatically changing data embedded within them using optimisation techniques to include \log_2 . As with the previous example (Section 5) we can use Table 1 as a guide. Naturally it leads to similar design choices. The next sections, where we describe the experiment which evolved \log_2 from the GNU C library sqrt, follow a similar structure to cbirt in Sections 5.1 to 5.3. However at the end of Section 6.4 we conclude that there are probably simpler implementations of \log_2 .

6.1 Evolving \log_2 Data Table via CMA-ES

As cbirt above, we use the existing glibc table driven implementation of the square root function sqrt and mutate the constant values in sqrt’s table to give a table driven implementation of the logarithm to base 2 function (\log_2).

6.2 \log_2 Manual Changes

As mentioned in Section 5.1, we are primarily concerned with adjusting data values but some changes to the existing powerPC sqrt code, Appendix A, were made by hand so that it could support \log_2 :

- Various powerPC optimisations were disabled.

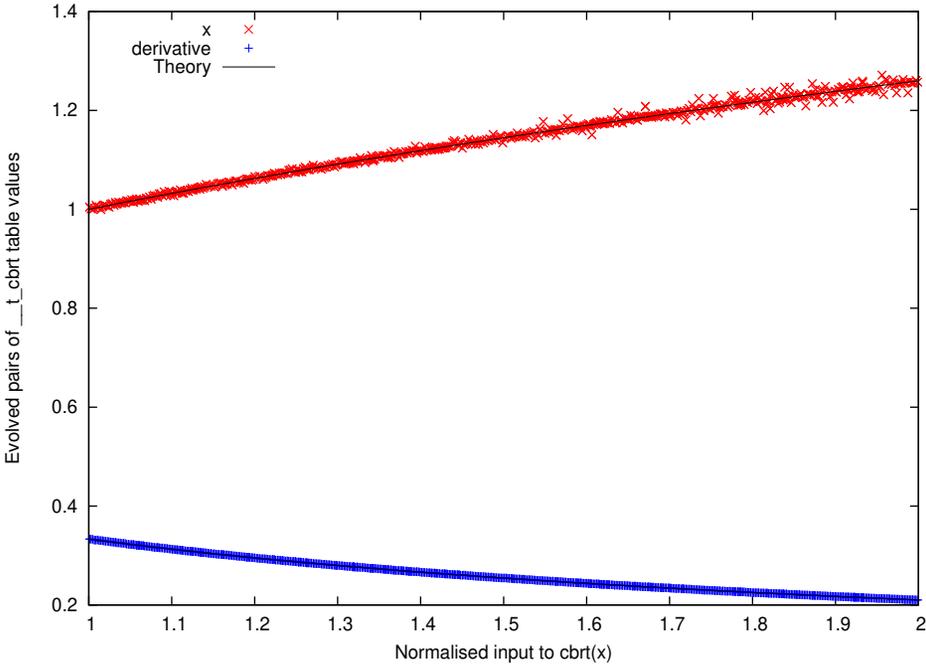


Fig. 10. 512 $x, \text{derivative of } x$ pairs of numbers found by CMA-ES for `__t_cbrt`. Horizontal axis is the normalised argument of `cbrt` which corresponds to each x, x' pair in `__t_cbrt`.

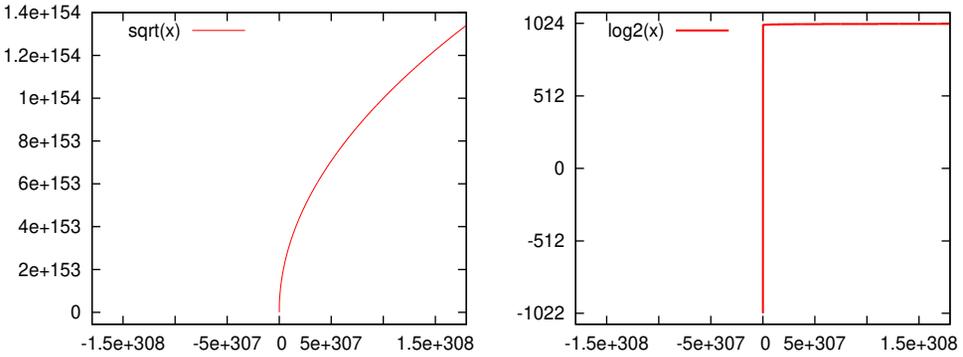


Fig. 11. Left: Double precision square root

Right: Double precision \log_2

- For simplicity the trap for negative numbers is implemented by failing an assert⁴.
- Since $\log_2(2^k \times x) = k + \log_2(x)$, `e_sqrt.c`'s division of the exponent part of double precision numbers by two is replaced by simply adding its value to the final result.
- As with `cbrt`, the top nine bits of the fractional part are used as the table index. Thus numbers in the range 1 to 2 are mapped onto the table's 512 entries. See also Figure 13.

⁴The GNU C library provides a sophisticated but complicated mechanism for raising exceptions, which would have to be used if the GI version of `log2` were to be incorporated back into `glibc`.

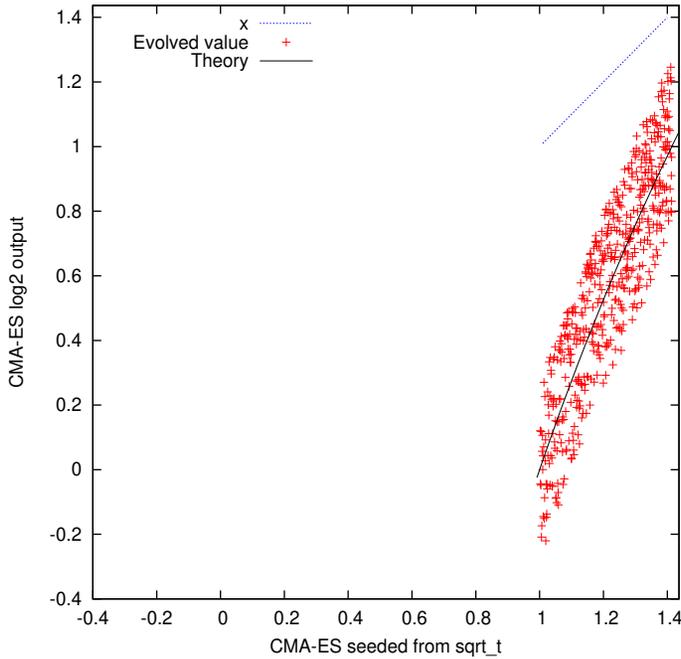


Fig. 12. Plot showing modification from sqrt table values (horizontal axis) to give corresponding log2 table value (vertical axis). 512 CMA-ES runs. (Diagonal blue line shows $x=y$, i.e. no change.)

- The Newton-Raphson objective function for sqrt (i.e. x^2) is replaced by 2^x . Since the derivative of $\log_2(x)$ can be calculated exactly from x (using the existing constant M_LOG2EI), it is not necessary to store it in the table or to estimate it or update the current estimate. Hence the table need only contain 512 values (rather than 512 pairs of values) and so the lines of code to maintain variables sy , $sy2$ and e can be removed. Similarly the constant $almost_half$ can also be removed.
- `e_sqrt.c`, see end of Section 5.1 and Appendix A, contains infrequently used recursive code to deal with tiny denormalised numbers (`denorm:`). This makes the number bigger by multiplying by 2^{108} . To undo this and so return the correct result, the new \log_2 code subtracts 108, i.e. $\log_2(2^{108})$, (in place of dividing by $\sqrt{2^{108}}$) to give the final result.

6.3 Evolving the log2 Data Table with CMA-ES

The `__t_sqrt` table contains 512 pairs of `float`. In `e_sqrt.c` each pair is the start point and starting derivative for the fixed iteration Newton-Raphson algorithm. As before, Section 5.2, only float precision is needed for the start points. Newton-Raphson will converge rapidly to double precision accuracy. By exploiting the IEEE754 floating point representation, the sqrt code converts all input x values into one of 512 possible starting bins. Each corresponding to an entry in `__t_sqrt`. For the new \log_2 code we do not need start values for the derivative, so the new table `__t_log2` becomes 512 `float`, which we are going to evolve using CMA-ES [17].

The top 256 `__t_sqrt` pairs correspond to numbers in the range 1 to 2. They become CMA-ES's initial (seed) value when it evolved the new table `__t_log2`. (CMA-ES is run 512 times, see Figure 12).

CMA-ES was set up to fill the `float` table one at a time. As with `cbirt`, Section 5.2, each value is initially set to either the corresponding value in `__t_sqrt` or the mean of two adjacent pairs. The

initial mutation step size used by CMA-ES was set to 3.0 times the standard deviation calculated from the first of each of the 512 pairs of numbers in `__t_sqrt`.

6.3.1 CMA-ES Parameters for \log_2 . The same CMA-ES parameters as were used for `cbirt`, Section 5.2.1, were used for `log2`, except the problem is one dimensional (i.e. N 1) rather than 2.

6.3.2 CMA-ES \log_2 Fitness Function. The `log2` fitness function follows that used for `cbirt`, Section 5.2.2. Each time CMA-ES proposes a value (N=1), it is converted into a `float` and loaded into `__t_log2` 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 `__t_log2` entry, the mid point and the top most value. Our `log2` function is called (using the updated `__t_log2`) for each and a sub-fitness value calculated with each of the three returned `double`. The sub-fitnesses are combined by adding them.

Each sub-fitness takes the output of our `log2`, and takes the absolute difference between this and the GNU `log2` library function. If they are the same, the sub-fitness is 0, otherwise it is positive. Since when our `log2` is working well, the differences are very small, they are re-scaled for CMA-ES. If the absolute difference is less than one, its (natural) log is taken, otherwise the absolute value is used. In both cases (i.e. as long as the difference is not zero), to prevent the sub-fitness being negative, log of the smallest feasible (i.e. encountered) non-zero difference is subtracted (i.e. a constant, 40.0, is added).

CMA-ES will stop when the fitness is zero. That is, the difference on all three test points is zero.

6.3.3 \log_2 Restart Strategy. Although a restart strategy was provided (as Section 5.2.3), in all 512 runs CMA-ES found a value for which all three test cases passed. Again the first reported solution was used. (Total run time 6 seconds).

6.4 Testing the Evolved \log_2 Function

The `float` values found by CMA-ES are shown in Figure 13. Figure 13 makes plain, that for a function as smooth as logarithm, no great care is needed in starting Newton-Raphson and more-or-less any reasonable value would do. Nonetheless we will continue to show that our evolved table driven version of the binary logarithm works as well as the GNU C library's `log2`. The `glibc-2.27` powerPC IEEE754 table-based `double sqrt` function is supposed to produce answers within one bit of the correct solution. On 1 536 tests of large integers ($\approx 10^{16}$) designed to test each of the 512 bins 3 times (minimum, maximum and a randomly chosen point) our GI `log2` always came within `DBL_EPSILON` (i.e. $2.2 \cdot 10^{-16}$) of the correct answer.

As well as the large positive integer tests mentioned in the previous paragraph, our `log2` was tested with 5 120 random numbers uniformly distributed between 1 and 2 (the largest deviation was in the least significant part of IEEE 754 double precision corresponding to $4.44 \cdot 10^{-16}$). It was also tested on 5 120 random scientific notation numbers and 5 120 random 64 bit patterns. Half the random scientific notation numbers were negative and half positive. 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 our `log2` correctly returned NAN. In all other cases our `log2` returned a `double`, which when fed into the GNU C library `exp2` function gave its input exactly or gave a number within one bit of the closest value which could be inverted by `exp2` to yield the original value.

In order to calculate the next iteration, Newton-Raphson requires an objective function (Figure 3), we use `exp2` in the objective function. Naturally this limits the usefulness of the Newton-Raphson approach for `log2`. $f(x) = \log_2(x) = \log(x)/\log(2)$ so the derivative is $f'(x) = 1/(x \log(2))$ and so the reciprocal needed by Newton-Raphson is $1/f'(x) = x \log(2)$, which can be easily calculated.

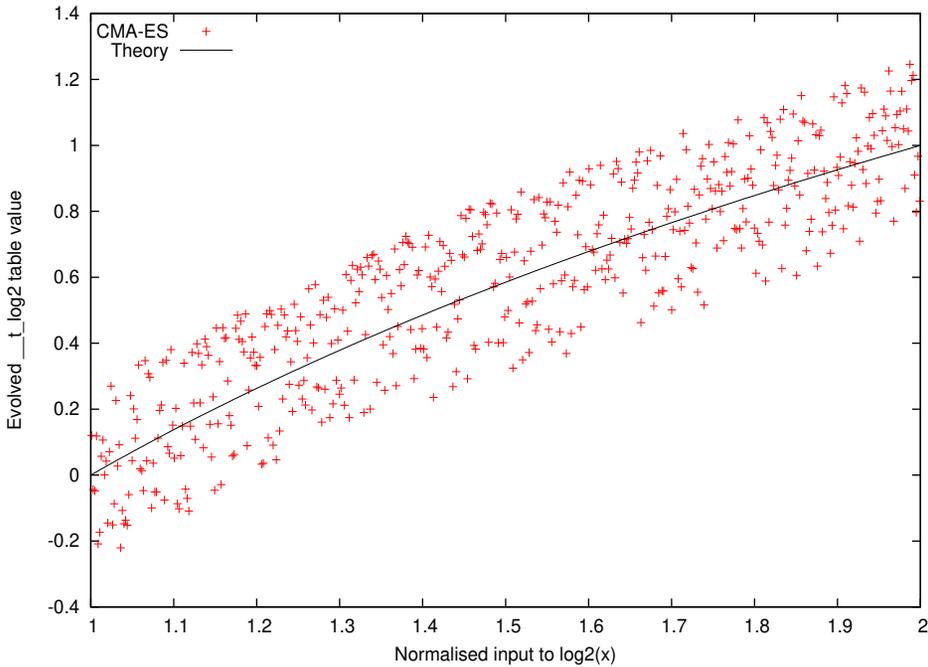


Fig. 13. 512 numbers found by CMA-ES for use as start points for Newton-Raphson three step calculation of the binary logarithm, i.e. the values in the float array `__t_log2`. The horizontal axis is the normalised input to `log2` corresponding to each value in `__t_log2`.

Whereas in `sqrt` (and our `cbt`) Newton-Raphson maintains and refines an estimate of $1/\text{derivative}$ at each step. Removing the code to do this greatly simplifies our `GI log2`. Further Figure 13 suggests that CMA-ES had almost no work to do and any reasonable estimate of `log2`, such as $x - 1$, would perhaps be sufficient. This would also allow the removal of the whole of the `__t_log2` table of start values and deleting even more of the `glibc sqrt` source code.

7 QUAKE AND $x^{-\frac{1}{2}}$

Quake was a first person shooter video game from the end of the second millenium C.E. It was designed to run on single user consumer electronics, such as personal computers and home video game consoles. For our purposes, it is noteworthy because it made extensive use of the reciprocal square root function ($x^{-\frac{1}{2}}$) in computer illumination calculations at a time when lack of hardware support made $x^{-\frac{1}{2}}$ expensive to compute. To produce high quality video displays many such calculations are needed. And yet there is very little time for calculation if the software is to achieve satisfactory interactive real-time response and refresh the user's video display at an acceptable rate. Quake solved this computational bottleneck by using a fast approximation to the reciprocal square root⁵. Today graphics cards (GPUs) provide hardware support for $x^{-\frac{1}{2}}$ specifically for interactive video games like Quake.

The GNU C library does not include `invsqrt` ($x^{-\frac{1}{2}}$). Although `invsqrt` can be readily calculated by taking the reciprocal of \sqrt{x} . Nonetheless even on powerful modern processors, typically division is

⁵https://en.wikipedia.org/wiki/Fast_inverse_square_root

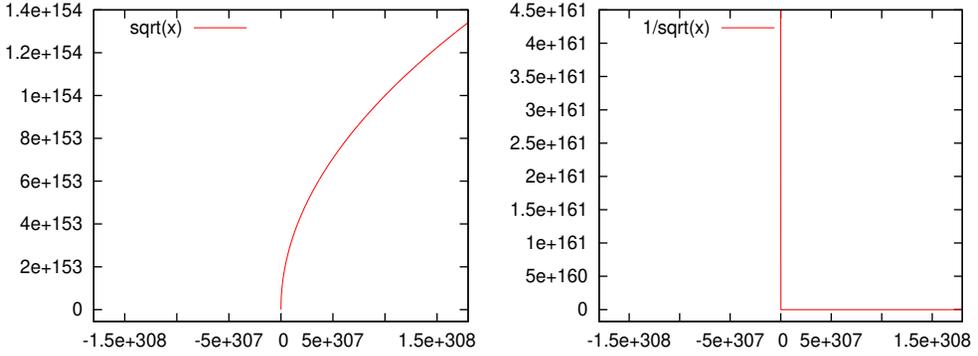


Fig. 14. Left: Double precision square root

Right: Double precision inverted square root

more expensive than multiplication. Not all processors provide hardware support for sqrt. Although hardware support for division is common, it need not be universal and may be provided by expensive software emulation. As computing becomes ubiquitous, with the internet-of-things and in particular mote computing, there will be a demand to run increasingly sophisticated algorithms (such as computer vision and machine learning) which may require vector normalisation on non-standard hardware lacking support for some mathematics functions and basic facilities we usually take for granted (such as a power supply). Hence there may be a demand for non-conventional mathematics implementations possibly providing unconventional trade-offs with energy consumption [40] or accuracy [59, 64] and a software based solution may be preferred to hardware circuits, such as [21].

As before (Sections 5 and 6) we can follow Table 1. We again start from `e_sqrt.c` (Appendix A) and use CMA-ES plus manual code changes to evolve a double precision table driven implementation of the reciprocal square root function $x^{-\frac{1}{2}}$ (Figure 14).

The following section (7.1) explains how to convert `e_sqrt.c` into $1/\sqrt{x}$ using manual changes (Section 7.2) and evolution (7.3). Section 7.4 shows the new function produces valid answers. We finish Section 7 with a comparison with Quake (Section 7.5) and discussion of possible other uses of $x^{-\frac{1}{2}}$ (Section 7.6).

7.1 Evolving `invsqrt` $x^{-\frac{1}{2}}$

7.1.1 Newton-Raphson Solves $x^{-\frac{1}{2}}$. The function whose root we want is $f(x) = \frac{1}{x^2} - \text{input}$. (I.e. what value of $x = \frac{1}{\sqrt{\text{input}}}$?) The derivative of $f(x)$ is $f'(x) = \frac{-2}{x^3}$.

The first iteration of Newton-Raphson (Section 4.1) is:

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)}$$

$$x_2 = x_1 + \left(\frac{1}{x_1^2} - \text{input} \right) \times \frac{x_1^3}{2}$$

$$x_2 = x_1 + \frac{(x_1 - x_1^3 \text{input})}{2}$$

Notice this formulation means we do not need to estimate $x^{-\frac{1}{2}}$ for the derivative and so we can avoid maintaining an estimate of its reciprocal but at the cost of three multiplications. Future GI work might seek optimal processor specific tradeoffs between memory needed to hold the Newton-Raphson estimate of the reciprocal of the derivative and processing time.

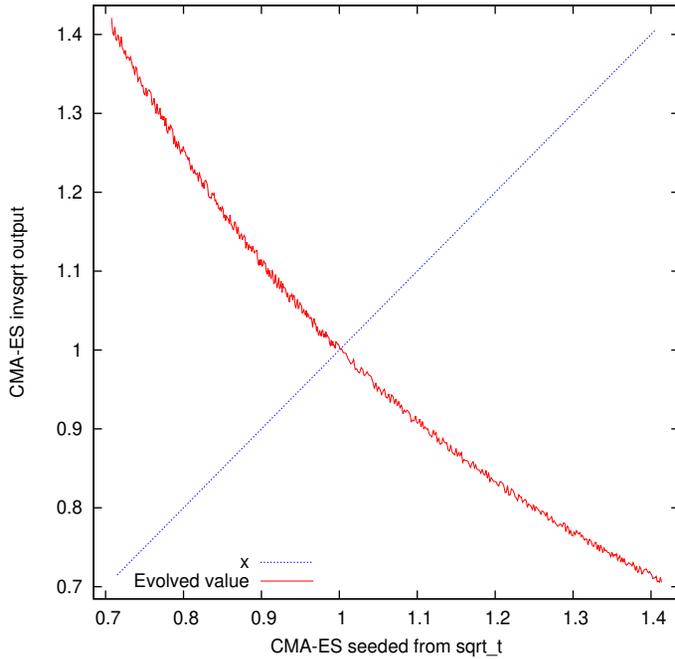


Fig. 15. Evolved change from sqrt table values (horizontal axis) to corresponding invsqrt table value (vertical axis). 512 successful CMA-ES runs. (Diagonal blue line shows $y=x$, i.e. no change.)

7.2 invsqrt Manual Changes

As with `cbirt` and `log2` (Sections 5.1 and 6.2) a small number of similar changes are needed before running evolution on the data table.

- The construction of the nine bit indexing operation is essentially unchanged but must take into account the table contains 512 float single values, not 512 pairs of float.
- The code to maintain the estimate of the reciprocal of the derivative was commented out.
- The new formula (Section 7.1.1) for the Newton-Raphson step is used (three times).
- The exponent part of the original floating point number must not only be divided by two (as in `e_sqrt.c`) but also must be negated. Since the IEEE 754 standard uses 11 unsigned bits to represent the exponent, this is accomplished by a right shift and then subtracting from the mid point ($1023 = 2^{(11-1)} - 1$).
- Dealing with denormalised double precision numbers is accomplished as in `e_sqrt.c` except the constant 2^{-54} is replaced by 2^{54} (see end of Appendix A).

7.3 CMA-ES Evolves $x^{-\frac{1}{2}}$ Data Table

The `__t_sqrt` table contains 512 pairs of float, corresponding to numbers in the range 0.5 to 2. The first of each pair was used as the start point when evolving the 512 float in the new table (see Figures 15 and 16).

We again use CMA-ES with initial values and the initial mutation step size based on `__t_sqrt` (Section 5.2).

7.3.1 invsqrt CMA-ES Parameters. The same CMA-ES parameters as were used for `log2`, Section 6.3.1, were reused for $x^{-\frac{1}{2}}$.

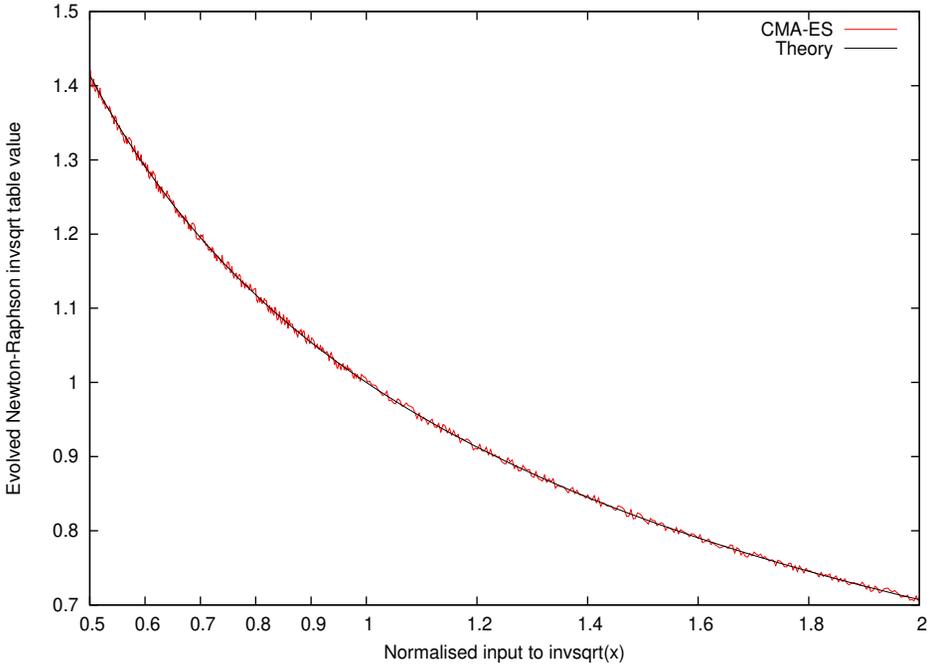


Fig. 16. 512 GI table values. Starts for 3 iterations of Newton-Raphson calculation of $x^{-\frac{1}{2}}$.

7.3.2 *invsqrt Fitness Function.* The CMA-ES used for $x^{-\frac{1}{2}}$ is similar to that used for *cbirt*, Section 5.2.2, and *log2*, Section 6.3.2, but as there are some difference in detail we next give a full description of it.

Each time CMA-ES proposes a value ($N=1$), it is converted from a `double` into a `float` and loaded into the table at the location that CMA-ES is currently trying to optimise. The fitness function uses three test `double` values in the range 0.5 to 2.0. These are: the lowest value for the table entry, the mid point and the top most value. The manually written code is called (using the updated table) for each and a sub-fitness value calculated with each of the three returned `double`. The sub-fitnesses are combined by adding them.

Each sub-fitness takes the output of manual code, and reverses it (i.e. squares it and then takes the reciprocal) and takes the difference between this and the corresponding test value. Of course if everything is working then they are the same. If they are the same, the sub-fitness is 0, otherwise it is positive. Since when *invsqrt* is working well, the differences are very small, they are re-scaled for CMA-ES. If the absolute difference is less than one, its natural 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 difference on all three test points is zero.

Since all the calculations are done as double precision approximations a certain degree of rounding inaccuracy can be expected. If the difference is really small, it may be treated as zero. To decide if the difference is small enough, the reverse operation is repeated for the `double` slightly bigger (i.e. larger = multiplied by $1+DBL_EPSILON$) and slightly smaller (i.e. smaller = divided by $1+DBL_EPSILON$) to give two new differences. The original answer is treated as close as possible to the right answer, i.e. fitness is zero, if either: 1) the original difference was zero or smaller than both the absolute difference of either smaller or larger. Or 2) the original difference is no more than $2 \times DBL_EPSILON$ and it lies between `d_smaller` and `d_larger`.

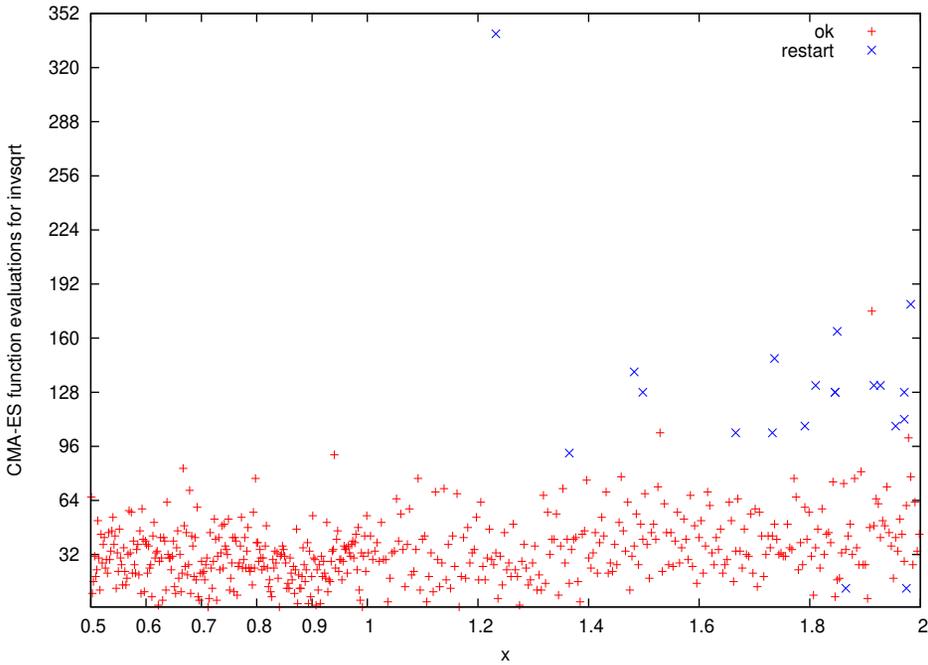


Fig. 17. CMA-ES finds it very easy to evolve 512 new start points for $x^{-\frac{1}{2}}$ when starting with data for $x^{\frac{1}{2}}$. (Mean 38.8 fitness evaluation.) There is no strong correlation with change to initial seed value (Figure 15). However all 20 of the runs which were restarted (x) are for $x > 1$.

7.3.3 *invsqrt* Avoiding Returning Negative Values. Notice all the steps in the fitness function (previous section) do not require comparison with an existing implementation. They take a purest approach of taking the inverse function ($f^{-1}(x) = x^{-2}$) and seeing how closely $f^{-1}(y)$ resembles the initial input. But notice $f^{-1}(x)$ is not monotonic. In particular $f^{-1}(-x) = f^{-1}(x)$. Thus from a mathematical perspective if y is a solution, then so too is $-y$. In one run evolution found such negative solutions to $1/\sqrt{x}$. Although mathematically sound, programming standards would not allow negative values. Therefore the fitness function was adjusted, so that negative numbers appear to be distant from $1/\sqrt{x}$, by adding $2x$ to the fitness objective. (Remember $0.5 \leq x < 2$ during testing and CMA-ES is minimising.)

7.3.4 *invsqrt* Restart Strategy. We use the cbtr restart strategy, Section 5.2.3. That is, if CMA-ES fails to find a suitable value for any of the 512 table values, it is run again with a different pseudo random seed but with the same initial starting position and mutation step size. In 494 of 512 cases CMA-ES found a suitable value in one run, but in 16 cases it was run twice, and in 2 it was run three times. To run CMA-ES 532 times took six seconds in total. Figures 17 and 18 shows the effort reported by CMA-ES for each evolutionary run.

7.4 Testing the Evolved *invsqrt*

Testing is based on that for cbtr, Section 5.3, and log2, Section 6.4. Similarly in all cases *invsqrt* produced a correct double precision answer.

On 1 536 tests of large integers ($\approx 10^{16}$) designed to test each of the 512 bins 3 times (minimum, maximum and a randomly chosen point) our GI *invsqrt* always came within a relative error of DBL_EPSILON (i.e. $2.22 \cdot 10^{-16}$) of the best possible answer.

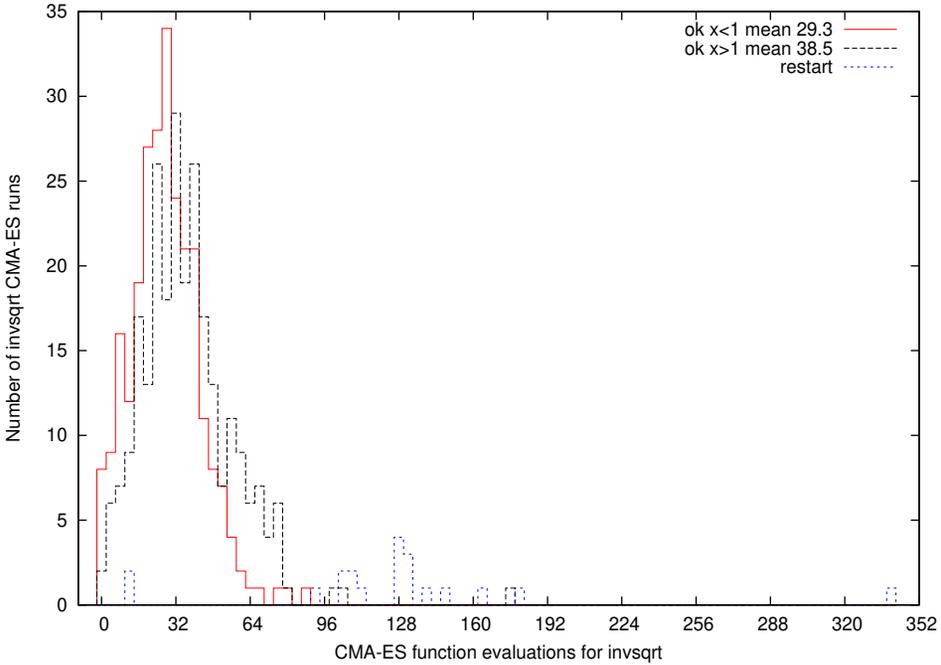


Fig. 18. Histogram (bin size 4) of number fitness evaluations per run for successful runs and for 20 runs which did not find an acceptable solution. Data as Figure 17. Excluding where CMA-ES was restarted (blue dashed line), search time for $1 < x < 2$ (black dashed line) is similar to $0.5 < x < 1$ (red solid line).

As well as ad-hoc testing, and the large positive integer tests mentioned in the previous paragraph, `invsqrt` was tested with 5 120 random numbers uniformly distributed between 1.0 and 2.0. (The largest relative deviation was 1.5 `DBL_EPSILON`.) It was also tested on 5 120 random scientific notation numbers and 5 120 random 64 bit patterns. Half the random scientific notation numbers were negative and half positive. Half were smaller than one and half larger. The exponent was chosen uniformly at random from the range 0 to $|\text{308}|$. In one case a random 64 bit pattern corresponded to NAN (Not-A-Number) and `invsqrt` correctly returned NAN. Again, in most cases `invsqrt` returned a double, which when squared and inverted was its input or within one bit of it. The maximum deviation from from the best possible answer was two in the fractional part, Figure 2 (i.e. a maximum relative error of `DBL_EPSILON`).

7.5 Comparison with Quake

The Quake (Section 7) fast inverted square root code, `InvSqrt`, was downloaded from stack overflow⁶ and subject to the same tests as the evolved `invsqrt` (Section 7.4). In terms of functionality Quake's `InvSqrt` is far worse:

- Quake `InvSqrt` gave the correct result to floating point precision in only 45 of the 16 903 tests. Often the result is up to 0.17% out. Unsurprisingly it never gave an answer correct to double precision accuracy.
- Quake `InvSqrt` does not deal with negative numbers. It may return `inf` but there are cases where it returns an incorrect floating point number.

⁶<https://stackoverflow.com/questions/268853/is-it-possible-to-write-quake-fast-invsqrt-function-in-c> See also `Q_rsq` in https://en.wikipedia.org/wiki/Fast_inverse_square_root (11 March 2019).

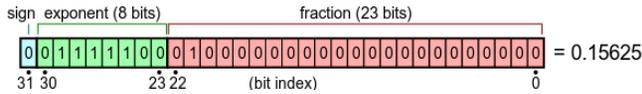


Fig. 19. Example of IEEE 754 single precision floating-point format (Wikipedia).

- Unsurprisingly Quake InvSqrt does not deal with double precision numbers outside the range of floating point precision (cf. Figures 2 and 19). It has odd behaviour for numbers smaller than $1.5 \cdot 10^{-37}$ and bigger than $3.3 \cdot 10^{38}$.

7.6 Other uses of invsqrt

Uses of the reciprocal square root function (invsqrt or $x^{-\frac{1}{2}}$) include computer graphics, artificial neural networks and machine learning, e.g. to normalise vectors.

8 DISCUSSION

There is some similarity with deep parameter tuning (also known as Deep Parameter Optimisation) [67][66][56][65][8][7][63][5]. Wu’s deep parameter tuning [66, 67] uses search based techniques to both expose and then tune values buried in existing source code. However Wu [67] deals with relatively few parameters, rather than more than five hundred. Also [67] seeks to tune existing functionality rather than, as here, to use data changes to create a new function, or to transplant existing functionality from one program to another [42][51][18].

Sections 2 and 3 and the previous paragraph have briefly covered the existing literature. They make clear that, apart from our own recent work [38] [35] [36] [28], the problem of automatic update of values embedded in existing software has been little studied.

Nowadays where much of software development is performed as continuous integration (CI), it may not be clear where the boundaries between development, bug-fixing and maintenance lies. Nonetheless the cost of software maintenance remains staggering. It is not uncommon for programs to contain, sometimes large numbers of, embedded constants. These may be critical to the software’s performance, and, although the problem has been known for a long time [45], there is as of yet very little research on automatically maintaining them.

Even parameters given by scientific measurement can be subject to change in just a few years [38]. In the case of one well used scientific package (ViennaRNA [35]), there are more than 50 000 integers which represent scientific knowledge. They had been deliberately corralled, to separate them in the source code, so that they could be updated as knowledge of the science increases. However, due to the expert manual effort involved, they had been updated by hand only once in several years, during which the software has been in routine use around the world. As an initial demonstration, we were able to show search based techniques could automatically update these compile time constants [38]. Andronescu et al. [3] have also automatically updated these parameters. They used constraint optimization. Nevertheless, our GI did significantly better [38, Fig. 4].

Section 4 expands parameter maintenance to the novel idea of creating new system software from existing functions via automated parameter tuning. Although we have used CMA-ES for our demonstration functions there are analytic approaches (note close agreement between CMA-ES and “Theory” in Figures 9, 10, 12, 13 and 16) which might have been used. Nonetheless these functions add further demonstrations of the power of evolutionary computation to optimise data embedded within existing programs. Sections 5 and 7 used CMA-ES to automatically adapt 1024 or 512 float constants, giving rise to cbrt and 1/sqrt, which do not currently exist in the C run time library (although cbrt is available in C++). Whilst in Section 6 our technique is demonstrated evolving data for log2. By considering $\sqrt[3]{x}$, \log_2 and $1/\text{sqrt}(x)$, we have shown our framework can be adapted

to provide new mathematics double functions [36],[27],[32] where there is an objective function, e.g. the inverse operation.⁷ We can view these as mostly a demonstration of what evolution can do and from a practical point of view perhaps it will prove most useful for porting existing functions to different hardware, possibly lacking direct mathematics support. For Newton-Raphson to be attractive, the function would have to be sufficiently non-linear and have a fast objective measure. Perhaps $\sqrt[3]{x}$, \sqrt{x} , $\psi \dots$ (with objective functions x^5 , x^7 , $x^{11} \dots$). The framework could be adapted to the trigonometric mathematics functions again by using the inverse function as the objective but to be useful the objective must either be computationally cheap or readily available (ideally both). However Taylor series expansion may be a practical alternative to Newton-Raphson. Also the availability of small (e.g. 4K bytes) of fast memory, e.g. within core, might suit this approach.

In very limited computing environments (e.g. tiny, ≈ 1 millimetre, processors such as smart dust, mote computing) it may be impossible to host GLIBC and there may be very limited electrical power for computation. There could still be realtime requirements, (e.g. track object, decide if it is near, close lens cap only if need be) which require a limited mathematics library. This limitation could take two forms. 1) Restrictions on inputs. E.g. angles are limited to -15° to $+15^\circ$ by the camera aperture. 2) Only a few combinations of functions are needed. This approach might be able to create an efficient implementation of such novel composite functions of no more than the required accuracy [12]. The inherent flexibility of software might mean that a novel bespoke software approach might be competitive with a custom hardware integrated circuit (ASIC) approach and still fit in a small read only memory (e.g. 4K bytes). If read only memory is very limited the tables might be reduced from 4K bytes to 256 bytes by using an 8 bit index into the table of start values (rather than 9 bits) and storing start values with 8 bit precision (rather than in 32 bit floats).

Previously [38] we have demonstrated using Genetic Improvement to adapt 50 000 parameters to new scientific knowledge may be possible but time consuming. Section 5 showed in less than five minutes evolution can adapt more than a thousand continuous values, whereas Sections 6.3.3 and 7.3.4 showed in a few seconds it can adapt several hundred continuous values.

Although testing cannot show the absence of bugs [16], software testing is indispensable for all software development [2, page 1979]. Testing is the *de facto* way of gaining confidence in software. It is notable that the the GNU C runtime library release kit comes with copious tests. The very directed testing in the fitness function hits the “edge cases”. That is, in addition to the mid point, it is precisely set to exactly test the extremities of each bin. After running CMA-ES, we have also used extensive testing to show that in normal operation (i.e. given a regular number) the functions return the correct double precision values. Excluding faults in the underlying hardware, we can be reasonably confident of our testing because once we exclude errors and special cases (e.g. negative inputs, `inf` and denormalised numbers), `e_sqrt.c` is just one linear sequence of instructions, which we have exercised thousands of times. As the functions of interest are well behaved in the range 0.5 to 2.0 and part of our testing covers the extrema of the individual bins as well as internal points, we can feel confident in the use of Newton-Raphson in normal operation. However, as mentioned above, e.g. in Section 6.2, we would want to make changes to the existing crude error reporting mechanism if the GI functions were used in the GNU C library.

The existing testing covers error conditions, `inf` and some denormalised numbers. However, as these are handled by separate code branches, before the GI code was used in a public general purpose library, the owners of the library would perhaps want to assure themselves that these side branches were also sufficiently tested. In particular since, the IEEE 754 double precision representation includes two types of infinity (positive and negative), many NAN and denormalised values, the

⁷It may be that the framework is sufficiently robust, that the task of evolving another function e.g. $\sqrt[3]{x}$, (possibly without the requirement to normalise the double precision numbers) might be posed as an exercise for talented students.

Table 2. Results

Start	Evolved	description	accuracy	search time
sqrt \rightarrow cbrt()	$\sqrt[3]{x}$	Section 5	double precision, i.e. $\leq 6.7 \cdot 10^{-16}$	270 seconds
sqrt \rightarrow log ₂ ()	$\log_2 x$	Section 6	double precision, i.e. $\leq 2.2 \cdot 10^{-16}$	6 seconds
sqrt \rightarrow invsqrt()	$x^{-1/2}$	Section 7	double precision, i.e. $\leq 2.2 \cdot 10^{-16}$	6 seconds

existing test cases for them ought to be extended. In particular since denormalised values are dealt with by recursive code, they should be tested more intensively. This is the only recursive code in `e_sqrt.c` and so, particularly in novel implementations, testing might uncover unexpected stack interactions, unrelated to the mathematical function itself. Additionally, e.g. following Markstein [43], it may be feasible to verify our evolved functions.

We have used optimisation (in the form of CMA-ES) on the data and traditional manual methods to change the source code. Future work might combine GI optimisation of data and source code. Several goals, even multi-objective simultaneous goals, might be considered. For example, run time and energy efficiency, or (particularly in mobile or ultra low resource systems) tradeoffs between speed, memory and code size. We have internally normalised `double` to the range 1.0 to 2.0. Future work could consider other internal normalisation ranges, e.g. 0.5 to 4.0.

9 CONCLUSIONS

This work hints, that in a world addicted to software, both automated data maintenance and data transplantaion could be essential new areas for optimisation research.

This approach combines minimal manual code changes (Sections 5.1, 6.2 and 7.2) and search. Starting from existing open source code, in a few seconds optimisation was able to find 512/1024 values to transform part of the GNU C library into a range of widely used mathematical functions. In all cases the GI functions produced a correct double precision answer (see Table 2).

ACKNOWLEDGMENTS

My thanks to Justyna Petke (UCL) and to our EuroGP [38] and GI @ GECCO 2019 [36] anonymous reviewers. Funded by EPSRC grant EP/M025853/1.

Code. Unix scripts and source code (including CMA-ES) are available via http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/gp-code/gi_cbrt.tar.gz and via a GitHub replication package https://github.com/oliver-krauss/Replication_GI_Division_Free_Division or doi:10.5281/zenodo.3755346

REFERENCES

- [1] B. J. Alexander and M. J. Grattton. 2009. Constructing an Optimisation Phase Using Grammatical Evolution. In *2009 IEEE Congress on Evolutionary Computation*, Andy Tyrrell (Ed.). IEEE Computational Intelligence Society, IEEE Press, Trondheim, Norway, 1209–1216. <http://dx.doi.org/10.1109/CEC.2009.4983083>
- [2] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John A. Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil McMinn. 2013. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software* 86, 8 (August 2013), 1978–2001. <http://dx.doi.org/10.1016/j.jss.2013.02.061>
- [3] Mirela Andronescu, Anne Condon, Holger H. Hoos, David H. Mathews, and Kevin P. Murphy. 2007. Efficient parameter estimation for RNA secondary structure prediction. *Bioinformatics* 23, 13 (2007), i19–i28. <http://dx.doi.org/10.1093/bioinformatics/btm223>
- [4] Earl T. Barr, Mark Harman, Yue Jia, Alexandru Marginean, and Justyna Petke. 2015. Automated Software Transplantation. In *International Symposium on Software Testing and Analysis, ISSTA 2015*, Tao Xie and Michal Young (Eds.). ACM, Baltimore, Maryland, USA, 257–269. <http://dx.doi.org/10.1145/2771783.2771796> ACM SIGSOFT Distinguished Paper Award.

- [5] Mahmoud A. Bokhari, Bobby R. Bruce, Brad Alexander, and Markus Wagner. 2017. Deep Parameter Optimisation on Android Smartphones for Energy Minimisation - A Tale of Woe and a Proof-of-Concept. In *GI-2017*, Justyna Petke, David R. White, W. B. Langdon, and Westley Weimer (Eds.). ACM, Berlin, 1501–1508. <http://dx.doi.org/10.1145/3067695.3082519>
- [6] Bobby R. Bruce. 2015. Energy Optimisation via Genetic Improvement A SBSE technique for a new era in Software Development. In *Genetic Improvement 2015 Workshop*, William B. Langdon, Justyna Petke, and David R. White (Eds.). ACM, Madrid, 819–820. <http://dx.doi.org/10.1145/2739482.2768420>
- [7] Bobby R. Bruce. 2018. *The Blind Software Engineer: Improving the Non-Functional Properties of Software by Means of Genetic Improvement*. Ph.D. Dissertation. Computer Science, University College, London, UK. http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/bruce_bobby_r_thesis.pdf
- [8] Bobby R. Bruce, Jonathan M. Aitken, and Justyna Petke. 2016. Deep Parameter Optimisation for Face Detection Using the Viola-Jones Algorithm in OpenCV. In *Proceedings of the 8th International Symposium on Search Based Software Engineering, SSBSE 2016 (LNCS, Vol. 9962)*, Federica Sarro and Kalyanmoy Deb (Eds.). Springer, Raleigh, North Carolina, USA, 238–243. http://dx.doi.org/10.1007/978-3-319-47106-8_18
- [9] Nathan Burles, Edward Bowles, Alexander E. I. Brownlee, Zoltan A. Kocsis, Jerry Swan, and Nadarajan Veerapen. 2015. Object-Oriented Genetic Improvement for Improved Energy Consumption in Google Guava. In *SSBSE (LNCS, Vol. 9275)*, Yvan Labiche and Marcio Barros (Eds.). Springer, Bergamo, Italy, 255–261. http://dx.doi.org/10.1007/978-3-319-22183-0_20
- [10] Simon Butler. 2015. *Analysing Java Identifier Names*. Ph.D. Dissertation. Open University, UK. <http://oro.open.ac.uk/46653/>
- [11] L. Cao, H. Sihler, U. Platt, and E. Guthel. 2014. Numerical analysis of the chemical kinetic mechanisms of ozone depletion and halogen release in the polar troposphere. *Atmospheric Chemistry and Physics* 14, 7 (2014), 3771–3787. <http://dx.doi.org/10.5194/acp-14-3771-2014>
- [12] Milan Ceska, Jiri Matyas, Vojtech Mrazek, Lukas Sekanina, Zdenek Vasicek, and Tomas Vojnar. 2017. Approximating Complex Arithmetic Circuits with Formal Error Guarantees: 32-bit Multipliers Accomplished. In *Proceedings of 36th IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, Iris Bahar and Sri Parameswaran (Eds.). Institute of Electrical and Electronics Engineers, Irvine, CA, USA, 416–423. <http://dx.doi.org/10.1109/ICCAD.2017.8203807>
- [13] David Peter Alfred Corney. 2002. *Intelligent Analysis of Small Data Sets for Food Design*. Ph.D. Dissertation. University College, London. <https://discovery.ucl.ac.uk/id/eprint/10099629>
- [14] Fabricio Gomes de Freitas and Jerffeson Teixeira de Souza. 2011. Ten Years of Search Based Software Engineering: A Bibliometric Analysis. In *Third International Symposium on Search based Software Engineering (SSBSE 2011) (LNCS, Vol. 6956)*, Myra B. Cohen and Mel O Cinneide (Eds.). Springer, Szeged, Hungary, 18–32. http://dx.doi.org/10.1007/978-3-642-23716-4_5
- [15] Sayed Mehdi Hejazi Dehaghani and Nafiseh Hajrahimi. 2013. Which Factors Affect Software Projects Maintenance Cost More? *Acta Informatica Medica* 21, 1 (Mar 2013), 63–66. <http://dx.doi.org/10.5455/AIM.2012.21.63-66>
- [16] E. W. Dijkstra. 1969. “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.). NATO, Scientific Affairs Division, Brussels, Rome, Italy, Chapter 3.1, 16. <http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1969.PDF>
- [17] Nikolaus Hansen and Andreas Ostermeier. 2001. Completely Derandomized Self-Adaptation in Evolution Strategies. *Evolutionary Computation* 9, 2 (Summer 2001), 159–195. <http://dx.doi.org/10.1162/106365601750190398>
- [18] Mark Harman, Yue Jia, and William B. Langdon. 2014. 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 (LNCS, Vol. 8636)*, Claire Le Goues and Shin Yoo (Eds.). Springer, Fortaleza, Brazil, 247–252. http://dx.doi.org/10.1007/978-3-319-09940-8_20 Winner SSBSE 2014 Challenge Track.
- [19] Mark Harman and Bryan F. Jones. 2001. Search Based Software Engineering. *Information and Software Technology* 43, 14 (Dec. 2001), 833–839. [http://dx.doi.org/10.1016/S0950-5849\(01\)00189-6](http://dx.doi.org/10.1016/S0950-5849(01)00189-6)
- [20] Yue Jia, Mark Harman, William B. Langdon, and Alexandru Marginean. 2015. Grow and Serve: Growing Django Citation Services Using SBSE. In *SSBSE 2015 Challenge Track (LNCS, Vol. 9275)*, Shin Yoo and Leandro Minku (Eds.). Springer, Bergamo, Italy, 269–275. http://dx.doi.org/10.1007/978-3-319-22183-0_22
- [21] John R. Koza, Forrest H Bennett III, Jason Lohn, Frank Dunlap, Martin A. Keane, and David Andre. 1997. Automated Synthesis of Computational Circuits Using Genetic Programming. In *Proceedings of the 1997 IEEE International Conference on Evolutionary Computation*. IEEE Press, Indianapolis, 447–452. <http://dx.doi.org/10.1109/ICEC.1997.592353>
- [22] Oliver Krauss and W. B. Langdon. 2020. Automatically Evolving Lookup Tables for Function Approximation. In *EuroGP 2020: Proceedings of the 23rd European Conference on Genetic Programming (LNCS, Vol. 12101)*, Ting Hu, Nuno Lourenco, and Eric Medvet (Eds.). Springer Verlag, Seville, Spain, 84–100. http://dx.doi.org/10.1007/978-3-030-44094-7_6
- [23] W. B. Langdon. 2012. Genetic Improvement of Programs. In *18th International Conference on Soft Computing, MENDEL 2012 (2nd ed.)*, Radomil Matoušek (Ed.). Brno University of Technology, Brno, Czech Republic. <http://www.cs.ucl.ac>

- uk/staff/W.Langdon/ftp/papers/Langdon_2012_mendel.pdf Invited keynote.
- [24] William B. Langdon. 2014. Genetic Improvement of Programs. In *16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2014)*, Franz Winkler, Viorel Negru, Tetsuo Ida, Tudor Jelebean, Dana Petcu, Stephen Watt, and Daniela Zaharie (Eds.). IEEE, Timisoara, 14–19. <http://dx.doi.org/10.1109/SYNASC.2014.10> Keynote.
- [25] W. B. Langdon. 2015. Genetic Improvement of Software for Multiple Objectives. In *SSBSE (LNCS, Vol. 9275)*, Yvan Labiche and Marcio Barros (Eds.). Springer, Bergamo, Italy, 12–28. http://dx.doi.org/10.1007/978-3-319-22183-0_2 Invited keynote.
- [26] William B. Langdon. 2015. Genetically Improved Software. In *Handbook of Genetic Programming Applications*, Amir H. Gandomi, Amir H. Alavi, and Conor Ryan (Eds.). Springer, Chapter 8, 181–220. http://dx.doi.org/10.1007/978-3-319-20883-1_8
- [27] W. B. Langdon. 2018. *Evolving Square Root into Binary Logarithm*. Technical Report RN/18/05. University College, London, London, UK. http://www.cs.ucl.ac.uk/fileadmin/UCL-CS/research/Research_Notes/RN_18_05.pdf
- [28] W. B. Langdon. 2019. Genetic Improvement of Data gives double precision invsqrt. In *7th edition of GI @ GECCO 2019*, Brad Alexander, Saemundur O. Haraldsson, Markus Wagner, and John R. Woodward (Eds.). ACM, Prague, Czech Republic, 1709–1714. <http://dx.doi.org/10.1145/3319619.3326800>
- [29] W. B. Langdon and M. Harman. 2010. Evolving a CUDA Kernel from an nVidia Template. In *2010 IEEE World Congress on Computational Intelligence*, Pilar Sobrevilla (Ed.). IEEE, Barcelona, 2376–2383. <http://dx.doi.org/10.1109/CEC.2010.5585922>
- [30] William B. Langdon and Mark Harman. 2015. Grow and Graft a better CUDA pknotsRG for RNA pseudoknot free energy calculation. In *Genetic Improvement 2015 Workshop*, William B. Langdon, Justyna Petke, and David R. White (Eds.). ACM, Madrid, 805–810. <http://dx.doi.org/10.1145/2739482.2768418>
- [31] William B. Langdon and Mark Harman. 2015. Optimising Existing Software with Genetic Programming. *IEEE Transactions on Evolutionary Computation* 19, 1 (Feb. 2015), 118–135. <http://dx.doi.org/10.1109/TEVC.2013.2281544>
- [32] William B. Langdon and Oliver Krauss. 2020. Evolving sqrt into 1/x via Software Data Maintenance. In *9th edition of GI @ GECCO 2020*, Brad Alexander, Alexander (Sandy) Brownlee, Saemundur O. Haraldsson, Markus Wagner, and John R. Woodward (Eds.). ACM, Internet, 1928–1936. <http://dx.doi.org/10.1145/3377929.3398110>
- [33] William B. Langdon, Brian Yee Hong Lam, Marc Modat, Justyna Petke, and Mark Harman. 2017. Genetic Improvement of GPU Software. *Genetic Programming and Evolvable Machines* 18, 1 (March 2017), 5–44. <http://dx.doi.org/10.1007/s10710-016-9273-9>
- [34] William B. Langdon and Justyna Petke. 2015. Software is Not Fragile. In *Complex Systems Digital Campus E-conference, CS-DC'15 (Proceedings in Complexity)*, Pierre Parrend, Paul Bourguine, and Pierre Collet (Eds.). Springer, 203–211. http://dx.doi.org/10.1007/978-3-319-45901-1_24 Invited talk.
- [35] William B. Langdon and Justyna Petke. 2018. Evolving Better Software Parameters. In *SSBSE 2018 Hot off the Press Track (LNCS, Vol. 11036)*, Thelma Elita Colanzi and Phil McMinn (Eds.). Springer, Montpellier, France, 363–369. http://dx.doi.org/10.1007/978-3-319-99241-9_22
- [36] W. B. Langdon and Justyna Petke. 2019. Genetic Improvement of Data gives Binary Logarithm from sqrt. In *GECCO '19: Proceedings of the Genetic and Evolutionary Computation Conference Companion*, Richard Allmendinger et al. (Eds.). ACM, Prague, Czech Republic, 413–414. <http://dx.doi.org/10.1145/3319619.3321954>
- [37] William B. Langdon, Justyna Petke, and Bobby R. Bruce. 2016. Optimising Quantisation Noise in Energy Measurement. In *14th International Conference on Parallel Problem Solving from Nature (LNCS, Vol. 9921)*, Julia Handl, Emma Hart, Peter R. Lewis, Manuel Lopez-Ibanez, Gabriela Ochoa, and Ben Paechter (Eds.). Springer, Edinburgh, 249–259. http://dx.doi.org/10.1007/978-3-319-45823-6_23
- [38] William B. Langdon, Justyna Petke, and Ronny Lorenz. 2018. Evolving better RNAfold structure prediction. In *EuroGP 2018: Proceedings of the 21st European Conference on Genetic Programming (LNCS, Vol. 10781)*, Mauro Castelli, Lukas Sekanina, and Mengjie Zhang (Eds.). Springer Verlag, Parma, Italy, 220–236. http://dx.doi.org/10.1007/978-3-319-77553-1_14
- [39] Ronny Lorenz, Stephan H. Bernhart, Christian Höner zu Siederdisen, Hakim Tafer, Christoph Flamm, Peter F. Stadler, and Ivo L. Hofacker. 2011. ViennaRNA Package 2.0. *Algorithms for Molecular Biology* 6, 1 (2011). <http://dx.doi.org/10.1186/1748-7188-6-26>
- [40] Jie Lu, Hongyang Jia, Naveen Verma, and Niraj K. Jha. 2018. Genetic Programming for Energy-Efficient and Energy-Scalable Approximate Feature Computation in Embedded Inference Systems. *IEEE Trans. Comput.* 67, 2 (Feb. 2018), 222–236. <http://dx.doi.org/10.1109/TC.2017.2738642>
- [41] Yanxin Lu, Swarat Chaudhuri, Chris Jermaine, and David Melski. 2018. Program Splicing. In *40th International Conference on Software Engineering*, Marsha Chechik and Mark Harman (Eds.). ACM, Gothenburg, Sweden, 338–349. <http://dx.doi.org/10.1145/3180155.3180190>

- [42] Alexandru Marginean, Earl T. Barr, Mark Harman, and Yue Jia. 2015. Automated Transplantation of Call Graph and Layout Features into Kate. In *SSBSE (LNCS, Vol. 9275)*, Yvan Labiche and Marcio Barros (Eds.). Springer, Bergamo, Italy, 262–268. http://dx.doi.org/10.1007/978-3-319-22183-0_21
- [43] P. W. Markstein. 1990. Computation of elementary functions on the IBM RISC System/6000 processor. *IBM Journal of Research and Development* 34, 1 (Jan 1990), 111–119. <http://dx.doi.org/10.1147/rd.341.0111>
- [44] Petr Marounek. 2012. Simplified approach to effort estimation in software maintenance. *Journal of Systems Integration* 3, 3 (2012), 51–63. <http://www.si-journal.org/index.php/JSI/article/view/123/99>
- [45] Roger J. Martin and Wilma M. Osborne. 1983. *Guidance on software maintenance*. NBS Special Publication 500-106. National Bureau of Standards, Department of Commerce, Washington DC, USA. <http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nbsspecialpublication500-106.pdf>
- [46] Michael Mohan and Des Greer. 2018. A survey of search-based refactoring for software maintenance. *Journal of Software Engineering Research and Development* 6, 3 (7 February 2018). <http://dx.doi.org/10.1186/s40411-018-0046-4>
- [47] Michael Orlov and Moshe Sipper. 2011. Flight of the FINCH through the Java Wilderness. *IEEE Transactions on Evolutionary Computation* 15, 2 (April 2011), 166–182. <http://dx.doi.org/10.1109/TEVC.2010.2052622>
- [48] Justyna Petke. 2015. Constraints: The Future of Combinatorial Interaction Testing. In *2015 IEEE/ACM 8th International Workshop on Search-Based Software Testing*. Florence, 17–18. <http://dx.doi.org/doi:10.1109/SBST.2015.11>
- [49] Justyna Petke. 2017. Preface to the Special Issue on Genetic Improvement. *Genetic Programming and Evolvable Machines* 18, 1 (March 2017), 3–4. <http://dx.doi.org/10.1007/s10710-016-9280-x> Editorial Note.
- [50] Justyna Petke, Saemundur O. Haraldsson, Mark Harman, William B. Langdon, David R. White, and John R. Woodward. 2018. Genetic Improvement of Software: a Comprehensive Survey. *IEEE Transactions on Evolutionary Computation* 22, 3 (June 2018), 415–432. <http://dx.doi.org/doi:10.1109/TEVC.2017.2693219>
- [51] Justyna Petke, Mark Harman, William B. Langdon, and Westley Weimer. 2014. Using Genetic Improvement and Code Transplants to Specialise a C++ Program to a Problem Class. In *17th European Conference on Genetic Programming (LNCS, Vol. 8599)*, Miguel Nicolau, Krzysztof Krawiec, Malcolm I. Heywood, Mauro Castelli, Pablo Garcia-Sanchez, Juan J. Merelo, Victor M. Rivas Santos, and Kevin Sim (Eds.). Springer, Granada, Spain, 137–149. http://dx.doi.org/10.1007/978-3-662-44303-3_12
- [52] Eric Schulte, Jonathan Dorn, Stephen Harding, Stephanie Forrest, and Westley Weimer. 2014. 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*. ACM, Salt Lake City, Utah, USA, 639–652. <http://dx.doi.org/10.1145/2541940.2541980>
- [53] Eric Schulte, Stephanie Forrest, and Westley Weimer. 2010. Automated Program Repair through the Evolution of Assembly Code. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*. ACM, Antwerp, 313–316. <http://dx.doi.org/10.1145/1858996.1859059>
- [54] Eric Schulte, Westley Weimer, and Stephanie Forrest. 2015. Repairing COTS Router Firmware without Access to Source Code or Test Suites: A Case Study in Evolutionary Software Repair. In *Genetic Improvement 2015 Workshop*, William B. Langdon, Justyna Petke, and David R. White (Eds.). ACM, Madrid, 847–854. <http://dx.doi.org/10.1145/2739482.2768427> Best Paper.
- [55] Madura A Shelton, Niels Samwel, Lejla Batina, Francesco Regazzoni, Markus Wagner, and Yuval Yarom. 2019. Rosita: Towards Automatic Elimination of Power-Analysis Leakage in Ciphers. arXiv. arXiv:1912.05183 [cs.CR] <https://arxiv.org/abs/1912.05183>
- [56] Jeongju Sohn, Seongmin Lee, and Shin Yoo. 2016. Amortised Deep Parameter Optimisation of GPGPU Work Group Size for OpenCV. In *Proceedings of the 8th International Symposium on Search Based Software Engineering, SSBSE 2016 (LNCS, Vol. 9962)*, Federica Sarro and Kalyanmoy Deb (Eds.). Springer, Raleigh, North Carolina, USA, 211–217. http://dx.doi.org/10.1007/978-3-319-47106-8_14
- [57] Hideyuki Takagi. 2001. Interactive Evolutionary Computation: Fusion of the Capabilities of EC Optimization and Human Evaluation. *Proc. IEEE* 89, 9 (Sept. 2001), 1275–1296. <http://dx.doi.org/10.1109/5.949485> Invited Paper.
- [58] Roberto Tiella and Mariano Ceccato. 2017. Automatic Generation of Opaque Constants Based on the K-Clique Problem for Resilient Data Obfuscation. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, Klagenfurt, Austria, 182–192. <http://dx.doi.org/10.1109/SANER.2017.7884620>
- [59] Zdenek Vasicek and Vojtech Mrazek. 2017. Trading between quality and non-functional properties of median filter in embedded systems. *Genetic Programming and Evolvable Machines* 18, 1 (March 2017), 45–82. <http://dx.doi.org/10.1007/s10710-016-9275-7>
- [60] David R. White. 2017. GI in No Time. In *GI-2017*, Justyna Petke, David R. White, W. B. Langdon, and Westley Weimer (Eds.). ACM, Berlin, 1549–1550. <http://dx.doi.org/doi:10.1145/3067695.3082515>
- [61] David R. White, Andrea Arcuri, and John A. Clark. 2011. Evolutionary Improvement of Programs. *IEEE Transactions on Evolutionary Computation* 15, 4 (Aug. 2011), 515–538. <http://dx.doi.org/10.1109/TEVC.2010.2083669>

- [62] David R. White, John Clark, Jeremy Jacob, and Simon M. Poulding. 2008. Searching for resource-efficient programs: low-power pseudorandom number generators. In *GECCO '08: Proceedings of the 10th annual conference on Genetic and evolutionary computation*, Maarten Keijzer et al. (Eds.). ACM, Atlanta, GA, USA, 1775–1782. <http://dx.doi.org/10.1145/1389095.1389437>
- [63] David R. White, Leonid Joffe, Edward Bowles, and Jerry Swan. 2017. Deep Parameter Tuning of Concurrent Divide and Conquer Algorithms in Akka. In *20th European Conference on the Applications of Evolutionary Computation (Lecture Notes in Computer Science, Vol. 10200)*, Giovanni Squillero and Kevin Sim (Eds.). Springer, Amsterdam, 35–48. http://dx.doi.org/10.1007/978-3-319-55792-2_3
- [64] Michal Wiggasz and Lukas Sekanina. 2018. Cooperative Coevolutionary Approximation in HOG-based Human Detection Embedded System. In *2018 IEEE Symposium Series on Computational Intelligence (SSCI)*. IEEE, Bangalore, India, 1313–1320. <http://dx.doi.org/10.1109/SSCI.2018.8628910>
- [65] John R. Woodward, Colin G. Johnson, and Alexander E. I. Brownlee. 2016. Connecting Automatic Parameter Tuning, Genetic Programming as a Hyper-heuristic, and Genetic Improvement Programming. In *GECCO '16 Companion: Proceedings of the Companion Publication of the 2016 Annual Conference on Genetic and Evolutionary Computation*. ACM, Denver, Colorado, USA, 1357–1358. <http://dx.doi.org/10.1145/2908961.2931728>
- [66] Fan Wu. 2017. *Mutation-Based Genetic Improvement of Software*. Ph.D. Dissertation. Department of Computer Science, University College, London, UK. http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/Thesis_Fan_v2.1.pdf
- [67] Fan Wu, Westley Weimer, Mark Harman, Yue Jia, and Jens Krinke. 2015. Deep Parameter Optimisation. In *GECCO '15: Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, Sara Silva et al. (Eds.). ACM, Madrid, 1375–1382. <http://dx.doi.org/10.1145/2739480.2754648>

A ORIGINAL CODE E_SQRT.C, FOR GLIBC 2.27 POWERPC SQRT

`__builtin_fma()` is a C compiler builtin library routine for efficiently doing multiply and add operations together.

`fesetenv_register()` and `fegetenv_register()` are system dependent macros to set and get the current floating-point environment.

`relax_fenv_state()` is a system dependent macro to set 1) the rounding mode to “round to nearest”; 2) the processor into IEEE mode; and 3) prevent exceptions from being raised for inexact results.

`__feraiseexcept()` is a system dependent macro which is part of the GNU C library’s exception handling. (Here dealing with attempts to take the square root of a negative double.)

`f_wash()` is a system dependent macro to 1) sets the appropriate Floating-point Status and Control Register bits for its parameter, 2) converts signaling NaN (sNaN) to the corresponding quiet NaN (qNaN), and 3) otherwise passes its parameter through unchanged (in particular, -0 and +0 stay as they were).

```
static const double almost_half = 0.5000000000000001; /* 0.5 + 2^-53 */
static const ieee_float_shape_type a_nan = { .word = 0x7fc00000 };
static const ieee_float_shape_type a_inf = { .word = 0x7f800000 };
static const float two108 = 3.245185536584267269e+32;
static const float twom54 = 5.551115123125782702e-17;
extern const float __t_sqrt[1024];
```

/* The method is based on a description in

Computation of elementary functions on the IBM RISC System/6000 processor,
P. W. Markstein, IBM J. Res. Develop, 34(1) 1990.

Basically, it consists of two interleaved Newton-Raphson approximations,
one to find the actual square root, and one to find its reciprocal
without the expense of a division operation. The tricky bit here
is the use of the POWER/PowerPC multiply-add operation to get the
required accuracy with high speed.

The argument reduction works by a combination of table lookup to
obtain the initial guesses, and some careful modification of the
generated guesses (which mostly runs on the integer unit, while the
Newton-Raphson is running on the FPU). */

```
double
__slow_ieee754_sqrt (double x)
{
    const float inf = a_inf.value;
    if (x > 0)
    {
        /* schedule the EXTRACT_WORDS to get separation between the store
        and the load. */
        ieee_double_shape_type ew_u;
        ieee_double_shape_type iw_u;
        ew_u.value = (x);
        if (x != inf)
    {
```

```

/* Variables named starting with 's' exist in the
   argument-reduced space, so that  $2 > sx \geq 0.5$ ,
    $1.41\dots > sg \geq 0.70\dots$ ,  $0.70\dots \geq sy > 0.35\dots$  .
   Variables named ending with 'i' are integer versions of
   floating-point values. */
double sx; /* The value of which we're trying to find the
   square root. */
double sg, g; /* Guess of the square root of x. */
double sd, d; /* Difference between the square of the guess and x. */
double sy; /* Estimate of  $1/2g$  (overestimated by 1ulp). */
double sy2; /*  $2*sy$  */
double e; /* Difference between  $y*g$  and  $1/2$  ( $se = e * fsy$ ). */
double shx; /*  $== sx * fsg$  */
double fsg; /*  $sg*fsg == g$ . */
fenv_t fe; /* Saved floating-point environment (stores rounding
   mode and whether the inexact exception is
   enabled). */
uint32_t xi0, xi1, sxi, fsgi;
const float *t_sqrt;

fe = fegetenv_register ();
/* complete the EXTRACT_WORDS (xi0,xi1,x) operation. */
xi0 = ew_u.parts.msw;
xi1 = ew_u.parts.lsw;
relax_fenv_state ();
sxi = (xi0 & 0x3fffffff) | 0x3fe00000;
/* schedule the INSERT_WORDS (sx, sxi, xi1) to get separation
   between the store and the load. */
iw_u.parts.msw = sxi;
iw_u.parts.lsw = xi1;
t_sqrt = __t_sqrt + (xi0 >> (52 - 32 - 8 - 1) & 0x3fe);
sg = t_sqrt[0];
sy = t_sqrt[1];
/* complete the INSERT_WORDS (sx, sxi, xi1) operation. */
sx = iw_u.value;

/* Here we have three Newton-Raphson iterations each of a
   division and a square root and the remainder of the
   argument reduction, all interleaved. */
sd = __builtin_fma (sg, sg, -sx);
fsgi = (xi0 + 0x40000000) >> 1 & 0x7ff00000;
sy2 = sy + sy;
sg = __builtin_fma (sy, sd, sg); /* 16-bit approximation to
   sqrt(sx). */

/* schedule the INSERT_WORDS (fsg, fsgi, 0) to get separation
   between the store and the load. */
INSERT_WORDS (fsg, fsgi, 0);

```

```

iw_u.parts.msw = fsgi;
iw_u.parts.lsw = (0);
e = __builtin_fma (sy, sg, -almost_half);
sd = __builtin_fma (sg, sg, -sx);
if ((xi0 & 0x7ff00000) == 0)
    goto denorm;
sy = __builtin_fma (e, sy2, sy);
sg = __builtin_fma (sy, sd, sg); /* 32-bit approximation to
sqrt(sx). */
sy2 = sy + sy;
/* complete the INSERT_WORDS (fsg, fsgi, 0) operation. */
fsg = iw_u.value;
e = __builtin_fma (sy, sg, -almost_half);
sd = __builtin_fma (sg, sg, -sx);
sy = __builtin_fma (e, sy2, sy);
shx = sx * fsg;
sg = __builtin_fma (sy, sd, sg); /* 64-bit approximation to
sqrt(sx), but perhaps
rounded incorrectly. */
sy2 = sy + sy;
g = sg * fsg;
e = __builtin_fma (sy, sg, -almost_half);
d = __builtin_fma (g, sg, -shx);
sy = __builtin_fma (e, sy2, sy);
fesetenv_register (fe);
return __builtin_fma (sy, d, g);
denorm:
/* For denormalised numbers, we normalise, calculate the
square root, and return an adjusted result. */
fesetenv_register (fe);
return __slow_ieee754_sqrt (x * two108) * twom54;
}
}
else if (x < 0)
{
/* For some reason, some PowerPC32 processors don't implement
FE_INVALID_SQRT. */
#ifdef FE_INVALID_SQRT
__feraiseexcept (FE_INVALID_SQRT);

fenv_union_t u = { .fenv = fegetenv_register () };
if ((u.l & FE_INVALID) == 0)
#endif
__feraiseexcept (FE_INVALID);
x = a_nan.value;
}
return f_wash (x);
}

```