Evolving sqrt into $1/x$ via Software Data Maintenance

W. B. Langdon
Department of Computer Science, UCL
London, UK
w.langdon@cs.ucl.ac.uk

Oliver Krauss
Johannes Kepler University,
AIST, University of Applied Sciences Upper Austria
Linz, Austria
oliver.krauss@fh-hagenberg.at

ABSTRACT

While most software automation research concentrates on programs' code, we have started investigating if Genetic Improvement (GI) of data can assist developers by automating aspects of the maintenance of parameters embedded in source code. We extend recent GI work on optimising compile time constants to give new functionality and describe the transformation of a GNU C library square root function into the double precision reciprocal function, drcp. Multiplying by $1/x$ (drcp) allows division free division without requiring the hardware to support division. The evolution (6 seconds) and indeed the GI dp division ($7.14 \pm 0.012 \text{ nS}$) are both surprisingly fast.

CCS CONCEPTS
- Software and its engineering → Search-based software engineering;

KEYWORDS
genetic programming, genetic improvement, Evolution Strategies, Covariance Matrix Adaptation - Evolution Strategy (CMA-ES), search based software engineering, SBSE, software maintenance of empirical constants, data transplantation, glibc, IoT, double precision (dp), reciprocal, drcp, rep, invert, inv

ACM Reference Format:

1 CONVENTIONAL DIVISION IS EXPENSIVE

On a modern 3.60 GHz desktop double precision multiplication takes a nanosecond, whereas double precision division takes about 4.0 times as long. For some systems which do not have floating point division in hardware, e.g. MMM [21], the ratio may be bigger. Indeed, even in some cases with hardware division, the ratio can be large. For example, on the ARM1176JZF-S (an ARM Vector Floating-Point coprocessor) the ratio is 14.5 [1, VFP 1-19]. Minimal systems, such as for internet-of-things (IoT) or ultra tiny mote computers [13] may not have the transistors or the power for conventional hardware division.

We use genetic improvement operating on data to show search can adapt existing embedded constants to repurpose existing open source C code to give a software double precision implementation of reciprocal (drcp). Our table driven drcp takes only 2 Kbytes (512 x 4bytes), which could be burnt into read-only memory (ROM) and as such is well within the reach of many mote processors. Not only can artificial evolution do this, but, when used in conjunction with multiply, we get a table driven software implementation of division which on our 3.60 GHz desktop ($7.14 \pm 0.012 \text{ nS}$) is only marginally slower than native 64 bit division.

We use the drcp function to show a new capability of GI in software maintenance. The source code itself is manually adapted, but the corresponding lookup table is automatically generated similarly to how software can be maintained when updated values are required, e.g. better approximations or changes in architecture from 32 bit to 64 bit.

The next section gives the background and shows, whilst Artificial Intelligence (AI) is increasingly used to maintain software source code, there is little research on maintaining numbers embedded in programs themselves. For this particular example (sqrt→drcp), Section 3 gives a brief introduction to iteratively finding a root of a mathematical equation using Sir Isaac Newton and Joseph Raphson’s iterative solver. A short introduction to CMA-ES is provided in Section 4. Whilst Section 5 describes the start point for this example: a GNU C mathematics library routine, sqrt (which uses Newton-Raphson), and how we use CMA-ES to evolve the data within it to give a new double precision reciprocal function, drcp. Note the evolved drcp does not use double precision division but can be used to replace it. The discussion (Section 6) shows that our GI division is accurate, and also considers possible future work. Finally, in Section 7 we conclude that for some low resource computers (such as for Internet of Things IoT [6] or approximate computing [55, 69]) the Genetic Improvement (GI) approach may help and that we have demonstrated evolutionary computing (EC) tools are opening up new approaches to automating software maintenance.

2 BACKGROUND: AI FOR SOFTWARE MAINTENANCE

Although computing is without doubt the success story of the second half of the twentieth century, at the beginning of the third millennium we are faced with an IT industry which remains labour intensive but not in the manufacture of the things but in looking after intangible IP (intellectual property), principally software. The lifetime cost of solid things, i.e. hardware, has fallen exponentially [54]. However, there has been no such dramatic change in
the cost of intangibles. Whilst production of hardware has been automated, software has not been automated and remains labour-intensive. These differences between computers and the software that runs on them has lead to very different maintenance regimes.

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

Initially, computer hardware was very diverse, demanding new software each time new hardware was commissioned. The advent of high-level languages and near monopolies in computer manufacture has lead to increasingly large volumes of software which can and has been reused. This led to “immortal software” [45, page 30] which has long outlived the computers it was originally developed for. Surprisingly, the consolidation into a few hardware and operating system environments has not lead to a similarly stable software industry. The economic pressure to be “first to market” has forced hardware innovation to be concentrated in a few computer hardware companies with fixed upgrade tram lines leading to users being “locked in” [57]. However, for most software, the race to be first continues to lead to great diversity in user programs.

The lack of automation in the software industry, online deployment and the economic necessity for software to be produced quickly, has lead to the rise of continuous deployment where new software is inflicted on the user as quickly as possible. Innovations in software production have fed the race to be first rather than increasing software quality. The lack of automation and the longevity of software have lead to software maintenance becoming the dominant cost of computing [14, 64].

Search based software engineering (SBSE) [25] uses AI tools to tackle software engineering problems. Increasingly SBSE is being used not just to find solutions to software problems, but to help software writers and maintainers by increasing the level of automation [23, 50],[2]. Rather than generating totally new programs [28, 29, 46], we are now seeing AI being used to automatically fix bugs [22, 47, 50] and improve existing software [71],[39, 61]. Genetic Improvement (GI) [59–62] has been used to optimise run time [39, 72], energy [10, 12, 66] and memory [73] efficiency, automatically importing functionality from one program into another [7, 49], growing new functionality and grafting it into another [24, 27, 31, 38] and indeed porting to new hardware [37]. GI on source code systems include GIN [70],[9] (Java), PyGGI [5] (multiple languages) and GISMOE [32] (C). Although GI can be applied to byte code [58], assembler [65] and indeed machine code [67], mostly GI has been applied to program source code, with little attention being paid to numbers embedded in software.

As well as external data to be processed, typically programs contain not just computer instructions, but also data. These may be float, int values or other types. For example, the GNU C library contains more than a million integers, see Figure 1 (also [42]). The numbers can be integral to the source code itself, but may also relate to the problem the program is solving, and as such may be subject to change just like the rest of the program’s environment [36], and so may need to be updated. The need to maintain data within software, as well as the code itself, has been recognized for a long time (Martin and Osborne, 1983 [52, Section 6.8]).

Although maintenance is the dominant cost of computing, a recent survey [53] starts by saying “a relatively small amount [of SBSE research] is related to software maintenance”, whilst de Freitas and de Souza [15] do not give a break down of the search based software engineering literature on software maintenance. Indeed there is little SBSE research on maintaining embedded numbers.

There is a little research on tuning of embedded parameters, Wu et al. [73]’s Deep Parameter Optimisation (DPO) work being the first example. They used DPO to adjust a few parameters to reduce runtime and memory. However, unlike DPO [11, 68, 73], we focus on adapting many numerical values to give better programs or indeed (as here) new functionality.

The ViennaRNA package [48] uses more than 50 000 free energy values. Recently we showed that genetic improvement can adapt these 50 000 int values to find a new program which on thousands of real examples gave predictions which were on average 11% more accurate [35].

We showed that evolution could update thousands of embedded constants to give new functionality [35, 43]. In [43] we argued that the technique could be widely applied and have applied it to generating log2 [44] and \( \frac{1}{\sqrt{x}} [33] \). We now use it to evolve a double precision division operator without division. We provide double precision division, \( x/y \), as \( x \times (1/y) \), i.e. \( x \times drcp(y) \). Where \( drcp(x) = 1/x \) is the double precision reciprocal or invert function.

---

1 In addition to 1029 floating point constants, the PowerPC double precision sqrt.c code contains 41 integer constants. E.g. logical values, error codes, array sizes and hexadecmal bit masks.
The next estimate, \( x_2 \), is given by updating \( x_1 \) by the error divided by the gradient (derivative) of \( f(x) \), \( f'(x) \)

\[
x_2 = x_1 - \frac{\text{error}}{f'(x_1)} = x_1 - \frac{f(x_1)}{f'(x_1)}
\]

Generalising, for the \((n + 1)\)th step

\[
x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}
\]

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

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

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

\[
\frac{f(x_n)}{f'(x_n)} = -\frac{f(x_n)x_n^2}{f'(x_n)} = -\left( x_n^{-1} - a \right)x_n^2 = -x_n + ax_n^2
\]  

(1)

Since \( x^2 \) is easily calculated, for drcp (unlike sqrt) we do not maintain an estimate for the reciprocal of the derivative. Therefore this part of e_sqrt.c was removed and \( x^2 \) was used instead (see Section 5.1).

### 4 COVARIANCE MATRIX ADAPATION - EVOLUTION STRATEGY (CMA-ES)

CMA-ES is an evolutionary algorithm used to solve n-dimensional continuous numerical problems. It has been shown to work for global and local optimization [20]. The algorithm evolves a population represented as a covariance matrix around a centroid. The centroid guides the evolutionary search by evolving a probability distribution. A standard deviation that is continuously and automatically updated during the run guides the mutation. The crossover happens by combining several individuals to new points in the covariance matrix. CMA-ES implicitly implements the crossover and mutation operators as they are directly tied to the core covariance matrix. CMA-ES does not require extensive parameter tuning, as all values for the operators are updated at regular intervals around the centroid [20].

CMA-ES can be provided with an initial configuration, such as the initial standard deviation and centroid. This only serves to speed up the algorithm to moving closer to an already expected or known optimum [20].

---

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

**Figure 3:** IEEE 754 Double-precision floating-point format. Notice sign (bit 63, light blue) is zero for positive numbers.
5 EVOLVING $1/x$ FROM GNU POWERPC $\sqrt{x}$

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

The GNU C library (release 31 Jan 2019 glibc-2.29) was used. It contains multiple implementations of the square root function (sqrt). As before [43], we selected some sysdeps/powerpc/fpu from the PowerPC implementations as it uses table lookup [51] (Figure 5). Again we adapted the GNU open source C code by hand and used Hansen’s CMA-ES [20] (see previous Section 4) to evolve the literals supplied by GNU for the square root function so that the code now calculates drcp.

5.1 Manual changes

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

- Negative numbers are caught before entering the main code. However, unlike sqrt, an error is not raised but instead $-x$ is passed to the main code and its result is negated. Thus, the main code does not deal with negative numbers.
- The construction of the nine bit indexing operation is essentially unchanged, but it must take into account that the table contains 512 floats not 512 pairs of floats (Figure 6).
- The code to maintain the estimate of the reciprocal of the derivative can be commented out.
- The new formula (Eq. 1) for the Newton-Raphson step is used three times.
- The GNU sqrt code deals with the exponent separately from the fractional (mantissa) part (Figure 3). To take the square root of the exponent, it is divided by two using a 1 bit shift right (left part Figure 5). For $x^{-1}$ the exponent part must be negated. Since the IEEE 754 standard uses 11 unsigned bits to represent the exponent, the new code subtracts it from the mid point ($1023 = 2^{11} - 1$) giving the exponent of the result (left part Figure 6). That is, the new code replaces masks and shift by masking and subtraction.
- The sqrt code deals with denormalised numbers (i.e. when the exponent part is zero, $x < 2^{-1023}$, see Figure 3) by multiplying by a large number and recursively calling itself and then adjusting the returned value appropriately. Except for using $2^{34}$, the new drcp is identical. It multiplies the tiny value $x$ by $2^{34}$. The drcp code is recursively called with the new (now normalised double precision value). The output will be $2^{-54}$ times too small and so the correct final value is obtained by multiplying by $2^{54}$. That is, the only code change is that the e_sqrt.e macros two108 = $2^{108}$ and two105 = $2^{-54}$ are both replaced by two54 = $2^{54}$.

5.2 Automatic changes to data table using CMA-ES

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

CMA-ES was downloaded from https://github.com/cma-es/c-maes/archive/master.zip. It was set up to fill the table of 512 floats one at a time. In all cases, the initial mutation step size used by CMA-ES was set to 3.0 times the standard deviation calculated from the 512 $x_1$ values in __t_sqrt.

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

5.2.2 CMA-ES Fitness function. Each time CMA-ES proposes a double value, it is converted into a float and loaded into the table at the location that CMA-ES is currently trying to optimise. For each of the 512 table entries, the fitness function uses three test points: the lowest value for the table entry, the mid point and the top most value. For simplicity, all the fitness test points are in the range 1.0 to 2.0. The invert function drcp that CMA-ES is trying to create is called (using the updated table) for each fitness test point and a sub-fitness value calculated with each of the three returned doubles. The sub-fitnesses are combined by adding them.

Sub-fitness is calculated by taking CMA-ES’s drcp output and inverting it (using 1.0/x). If the evolved value was correct, the answer would be the same as the test input value. (Effectively the fitness function is using metamorphic testing [8], which avoids having a test oracle which knows in advance the desired answer for every fitness test case.) Sub-fitness is based on the absolute difference between these. If they are the same or very close, the sub-fitness is 0. We define “very close” to mean the difference is less than either the difference calculated when inverting a number very slightly smaller than CMA-ES’s drcp’s output or when inverting a number very slightly bigger. “Slightly” meaning to the best double precision accuracy, i.e. multiplied or divided by $(1 + \text{DBL}\_\text{EPSILON}) = (1 + 2^{-52})$. (DBL_EPSILON in C is the minimal value which when added to 1.0 which results in a different double value.)

If the output from the evolved drcp is not close enough, the sub-fitness is positive. When drcp is working well the differences are very small, therefore they are re-scaled for CMA-ES, although this may not be essential [30]. If the absolute difference is less than one, its log is taken, otherwise the absolute value is used. However, in both cases, to prevent the sub-fitness being negative, log of the smallest feasible non-zero difference DBL_EPSILON is subtracted.

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

5.2.3 Restart Strategy. If CMA-ES fails to find a value for which all three test cases are ok, it is run again with the same initial starting position and mutation size, but a new pseudo random number seed. In 467 cases CMA-ES found a suitable value in one run, but in 39 of 512 cases it was run twice, and in 6 cases three CMA-ES runs were needed. This shows that the selected fitness function is robust, finding the global optimum in all cases. However we still require some restarts due to the stochastic nature of the approach.
Evolving sqrt into $1/x$ via Software Data Maintenance

Figure 4: Left: Double precision square root, $\sqrt{x}$, $\sqrt{x}$. Right: Double precision reciprocal, drcp(x), $1/x$. Note in both plots the horizontal axis covers the full range of double precision numbers and the two functions give very different outputs (y-axis).

Figure 5: Left: e_sqrt.c 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 (mantissa) part to give nine bit index. Right: index used with float __t_table containing 512 $x_1$ and $1/2x_1$ pairs of initial values for Newton-Raphson iterative solution of $\sqrt{x}$.

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

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

5.3 Testing the evolved drcp function

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

The evolved drcp was also tested with 5 120 random numbers uniformly distributed between 1 and 2 (the largest deviation was two), 5 120 random scientific notation numbers and 5 120 random 64 bit patterns. Half the random scientific notation numbers were

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

It took 6 seconds to run CMA-ES 563 times on one core of a 3.60GHz i7-4790 Intel desktop computer. The search effort is given in Figures 7 and 8. The values input to CMA-ES and those output by it are given in Figures 9 and 10.
negative and half positive. In absolute value, half were smaller than one and half larger. The exponent was chosen uniformly at random from the range $0$ to $|308|$.

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

6 DISCUSSION

6.1 Sufficient Testing?

Although it is well-known that testing cannot prove correctness [17], in keeping with the IT industry as a whole (indeed the GNU developers themselves, see Section 6.3), we have sought to demonstrate our evolved implementation using testing (see Section 5.3). Excluding tests for exceptions such as denormalised numbers and range errors, the code is straight through with no loops or branches and drcp (in the range of interest, 1.0 to 2.0) is monotonic and smooth. We have shown this main code yields correct double precision answers thousands of times, including covering all 512 data bins multiple times, including their edges. (Indeed a proof, following Markstein [51], might be possible.) We can thus be reasonably confident of the GI code in normal operation.

Although the testing has covered some special cases, it is noticeable that the GNU mathematics library developers have included almost as many tests for exceptions as for normal cases. We have been concerned primarily with normal operation and not attempted to use the glibc exception handling code. Therefore if our evolved drcp were to be included in glibc they might well want to satisfy themselves that non numeric inputs such as nan, +inf and -inf are also dealt with correctly. Similarly additional testing might be used when dealing with non-normalised numbers, cf. Section 5.1, especially as this is the only instance of recursive code.

6.2 Originality, Utility and Scope for Disruption of Software Engineering

As the literature review in Section 2 makes clear this is an under explored area and yet Evolutionary Computation (EC) can sometimes rapidly produce useful results (here with a run time of a few seconds). By working with software maintainers, EC-based AI data maintenance tools could make a significant dent in the software maintenance mountain.
Our evolved double precision division \( \frac{x}{y} = x \times \text{drcp}(y) \) is potentially competitive on processors where division is slow. In particular, if division is slower than multiplication by a ratio of more than 8, multiplication by the evolved double precision reciprocal, \( \text{drcp} \), will be faster than division. For example, there are some processors (see Section 1) where the time to do double precision division is more than 14\( \times \) the time to do double precision multiplication, e.g. ARM1176JZF-S [1]. On such hardware \( x \times \text{drcp}(y) \) would be faster than \( x/y \).

As before [43], we have only updated the read-only data table. Comments in the GNU PowerPC C source code make the point that a skilled software designer chose the interleaving of the instructions to maximise performance on the PowerPC. Since we desire to make our description as complete as possible for a scientific audience, cf. Section 5.1, we have gone into perhaps more detail than is needed and so perhaps given the impression that the Newton-Raphson code is more complex than it actually is. In practice, the GNU sqrt code can be easily adapted. Secondly, for speed, the code uses a data-driven approach for the Newton-Raphson derivative. Both of these were manual choices by the GNU programmers for the PowerPC. Future research might investigate using genetic improvement on the code in order to see if this is optimal on other computers, if the data-derivative approach should be used in drcp, or if GI can find other optimisations.

The bulk of software maintenance is routine and tedious, although in some cases it may require highly skilled experts [16, page 65]. In either case, the available human resources may already be stretched thin. Thus partial automation via AI can be useful, even (as is shown in [35]) for high skill requirements.

6.3 Future work:
GI Autoport Test Cases and Test Oracles

The GNU C library includes more than 6000 extensive test suites. These include the square root test cases, of 599 individual tests. Like other glibc math functions, they are used for complex numbers (i.e., with real and imaginary parts), different precisions (long double, double, float etc.) and inlined and noninlined code. Just concentrating upon testing \( \sqrt{x} \) (i.e. double), the tests are executed 1360 times (plus 1348 tests for errors and exceptions). Although AI has made considerable progress in generating test cases to exercise code [4, 19] usually this relies on implicit oracles [8, 26, 34, 56, 63] (such as: does the test cause the code to crash with a null pointer exception?), or approximate oracles such as: does the output include strings such as “error”, “problem” or “exception” [18, page 262]? We have used traditional manual testing to demonstrate our drcp, however, what if we were to use not just the existing implementation but the existing test suite and use it to test the new functionality? How much adaptation of the tests would be needed? How much of this could be automated? Would genetic improvement be able to evolve existing test cases and their test oracles? The little work on automatic test case porting [74] and the presence of thousands of test suites with hundreds tests and their test oracles makes using GI to transplant glibc test suites a tempting target for future research.

7 CONCLUSIONS

The cost of software maintenance is staggering. Although support tools are common place, it remains an essentially tedious error prone manual process with little existing evolutionary computing (EC) research. Indeed, although recognised since the early 1980’s, there is little research on automatic ways to maintain numeric values even though this is an important part of software maintenance. Figure 1 shows an example of important long lived software which contains a large number of embedded constants.

Most EC software engineering research has concentrated upon source code. Although we already have a few examples of GI programs in use and under regular software maintenance [40] [41] [22] [3, 50], there is a fear that some in the IT industry might be resistant to AI automated source code improvement. Since software developers care about their source code, potentially, by concentrating automatic updates on parameters within code, rather than the instructions, it may be that they will be more accepting of evolved artefacts.

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

Acknowledgements
We are grateful for the assistance of Justyna Petke, Roy Longbottom and our anonymous reviewers. Also we would like to acknowledge Wikipedia (Figures 2, 3, etc.) and thank Simon Tatham for PuTTY.

Funded by EPSRC GGGP and InfoTestSS grants EP/M025853/1 EP/P005888/1.

A replication package is available via DOI: https://doi.org/10.5281/zenodo.3755346 on GitHub https://github.com/oliver-krauss/Replication_GI_Division_Free_Division

REFERENCES


