Automatic Linear Correction of Rounding Errors for Newton's Method

Goktug Saatcioglu

May 10, 2018

Abstract

This project explores the application of the CENA method on Newton's method for one-dimensional zero finding through experimental evaluation. The study shows that, in general, the CENA method provides accuracy improvements over naive implementations but is not as good as doubling the working precision. Possible areas for the uses of the CENA method are also highlighted.

1 Introduction

Rounding errors are a "fact of life" when dealing with any computation in finite precision arithmetic. Famous examples of arithmetic errors include the many jokes inspiring Intel division bug [19] and the subtle accumulating rounding errors suddenly revealing itself in the Vancouver stock exchange [14]. More serious examples of rounding errors leading to catastrophic consequences include the explosion of the spacecraft Ariane 5 due to an unexpected conversion from a 64-bit floating point number to a 16-bit integer [13] and an error with computing time with both integer and floating point numbers leading to a failure in the Patriot missile system leading to the death of 28 people during the Gulf war [20]. By now it is clear that there is an imperative need to devise techniques to address such issues. Methods for dealing with either correcting such errors, detecting them or re-writing code to mitigate such issues, broadly speaking, falls under static, dynamic and error bound determining approaches. Static analysis generally uses abstract interpretation to provide sound approximations of the semantics of programs such as in [21] and [15]. On the other hand, dynamic approaches involve either running a program several times to estimate error or inserting extra code to analyze certain computations as can be seen in [8]. Work related to finding relative and absolute error bounds can be found in [3] and [18]. Correction Linéaire Automatique des Erreurs d'Arrondi (CENA) is a unique correcting method introduced by Langlois in [9] that, instead of approximating or analyzing a program, computes the forward error accumulated over the program's execution. This paper analyzes the effectiveness of the

CENA compensations on Newton's method for one-dimensional zero finding. In Section 2 the standard model of floating point numbers and arithmetic is introduced. Section 3 presents error free transformations and how they are computed. Then, Section 4 presents the CENA method and ties everything from the previous section together. The framework for CENA Newton's method is given in Section 5 along with experimental results on a specific type of polynomial. Finally, in Section 6 the flaws in the analysis are outlined along with an hollistic overview of CENA and Section 7 provides a conclusion.

2 Floating Point Arithmetic

2.1 The Standard Model of Representation

The IEEE Standard for Floating-Point Arithmetic [22], or by its more common name IEEE-754, is the current standard for working in finite precision arithmetic and "mandates that floating point operations be peformed as if the computation was done with infinite precision and then rounded." [2] Let $\mathbb{F} \subset \mathbb{R}$ be the set of floating point numbers with precision ϵ_M where ϵ_M is commonly referred to as "machine precision." A normalized number $\hat{x} \in \mathbb{F}$ is represented as

$$\hat{x} = \pm(1 + m) \times b^e$$

where $b = 2$ or $b = 10$ is the base, e (the exponent) is a signed integer and m is a fixed point value between 0 and 1. In single precision floating point numbers m is a 23-bit value and e is a 8-bit value while in double precision floating point numbers m is a 52-bit value and e is an 11-bit value. Taking into the account the signed bit, there are 32 bits for single and 64 bits for double precision numbers. Special values occur when $e = 0, m = 0$ which gives $+0$ and -0 depending on the sign bit, $e = 1 \ldots 1, m = 1$ which gives $+\infty$ and $-\infty$, and $e = 1 \ldots 1, m = 0$ which gives the value "not a number', also known as `NaN`. In general, operations that analytically lead to an indeterminate form, such as $\infty - \infty$, will produce `NaN`. Denormalzied (also known as subnormal) numbers $\tilde{x} \in \mathbb{F}$ in the form

$$\tilde{x} = \pm(0 + m) \times 2^e$$

allow the system to represent small numbers with zero exponent and non-zero mantissa by allowing for gradual underflow. Without loss of generality, this project will assume the most commonly used floating point numbers which are the normalized values with base $b = 2$. IEEE-754 also specifies other precisions and a nice summary table can be found in [15].

2.2 Rounding

Referring back to [2], rounding modes that are given in [22] are:

- round towards nearest,
- round towards $+\infty$,

- round towards $-\infty$, and
- round towards 0.

In the case of round towards nearest, the "tablemaker's dilemma" situation arises where a choice must be made on how to round a value that lies exactly in between two representable numbers. The two options for resolving the dilemma are:

- round to the nearest even, and
- round to the largest magnitude.

Combining everything together, let $x \in \mathbb{R}$ be some number and $\hat{x} \in \mathbb{F}$ be its rounded floating point representation. A standard way of expressing the transformation from x to $\hat{x}$ is

$$\hat{x} = fl(x) = x(1+\delta), \quad \text{with } |\delta| < \epsilon_M$$

where δ is the error associated with rounding and is zero if and only if x is exactly represntable. Again, without loss of generality, this project will assume the most commonly used rounding of round towards nearest with rounding direction towards the nearest even where $\epsilon_M = 2^{-24}$ for single-precision and $\epsilon_M = 2^{-53}$ for double-precision floating point representations.

2.3 The Standard Model of Arithmetic

Elementary floating point operations are defined by the set $\{+, -, \times, /\}$. For $\hat{x}, \hat{y} \in \mathbb{F}$ and $\circ \in \{+, -, \times, /\}$ let $z = \hat{x} \circ \hat{y}$ and $\hat{z} = fl(z) = fl(\hat{x} \circ \hat{y})$ (with $\hat{y} \neq 0$ if $\circ = /$). Then, along with the standard model of representation and rounding defined above, the standard model of arithmetic is given by

$$\hat{z} = fl(z) = fl(\hat{x} \circ \hat{y}) = (\hat{x} \circ \hat{y})(1+\delta_1), \quad \text{with } |\delta| \leq \epsilon_M, \quad \text{where } \circ \in \{+, -, \times, /\}$$

where δ is the the error associated with the rounding of elementary operations, or simply elementary rounding errors, assuming no overflow or underflow has occurred. [6] "The model says that the computed value of $\hat{x} \circ \hat{y}$ is "as good as" the rounded exact answer, in the sense that the relative error bound is the same in both cases." [6] Furthermore, the consideration of the errors associated with $\hat{x}$ and $\hat{y}$ due to different possible "lengths" of the values is not necessary due to the use of guard digits in the IEEE-754 standard. [6]

3 Error Free Transformations

3.1 Error Free Transormations for $+, -\times$

Re-writing the standard model of arithmetic gives us

$$\delta = (\hat{x} \circ \hat{y}) - fl(\hat{x} \circ \hat{y})$$

where δ is the elementary rounding error in the computation of $\hat{x} \circ \hat{y}$. "In particular, for $\circ \in \{+, -, \times\}$, the elementary rounding error" $\delta \in \mathbb{F}$ "and is computable using only the operations defined within F. [11] Thus, for any operation $\circ \in \{+, -\times\}$ and a pair of inputs $(\hat{x}, \hat{y}) \in \mathbb{F}^2$, any algorithm that produces an output pair $(fl(\hat{x} \circ \hat{y}), \delta) \in \mathbb{F}^2$ such that

$$\hat{x} \circ \hat{y} = fl(\hat{x} \circ \hat{y}) + \delta$$

is called an error free transformation, or EFT for short. [11] [17] The main advantage of using an EFT is that "no information is lost" regarding the value and error from an elementary floating point operation as long as the operation is addition, subtraction or multiplication.

Perhaps the most well known EFT is the summation algorithm by Knuth presented in [7], which is also valid for subtraction by flipping the sign bit of one of the inputs. Defining $\oplus, \ominus, \otimes$ as the floating point operations for $+, -, \times$ respectively, the `TwoSum` algorithm is given below and its Matlab implementation can be found in A.1.

Algorithm 3.1: KNUTH'S EFT ALGORITHM FOR SUMMATION( )

procedure TWOSUM(x, y)

$z \leftarrow x \oplus y$
$s \leftarrow z \ominus x$
$\delta \leftarrow (x \ominus (z \ominus s)) \oplus (y \ominus s)$
return (z, δ)

For the EFT of a multiplication it is first necessary to split the inputs into two parts such that a number $\hat{x} \in \mathbb{F}$ can be expressed as

$$\hat{x} = \hat{x}_h + \hat{x}_l \quad \text{with} \quad |\hat{x}_l| \leq |\hat{x}_h| \,.$$

If $\hat{x}$ has m (mantissa) with q bits, then define $r = \lceil q \rceil$. The splitting algorithm by Dekker defined in [4] will split $\hat{x}$ into the two parts as defined above where both parts will have at most $r-1$ non-zero bits. The `Split` algorithm is given below and its Matlab implementation can be found in A.2.

Algorithm 3.2: DEKKER'S SPLITTING ALGORITHM( )

procedure SPLIT(x, q)

$r \leftarrow \lceil q \rceil$
$z \leftarrow x \otimes (2^r + 1)$
$x_h \leftarrow z \ominus (z \ominus x)$
$x_l \leftarrow x \ominus x_h$
return (x_h, x_l)

Here a signed bit is used for the splitting which makes it possible to split an odd number of mantissa bits into two numbers of an even amount of bits. Using the splitting algorithm it is then possible to define Veltkamp's EFT for multiplication. [4] The `TwoProduct` algorithm is given below and its Matlab implementation can be found in A.3.

Algorithm 3.3: VELTKAMP'S MULTIPLICATION ALGORITHM( )

procedure TWOPRODUCT(x, y, q)

$z \leftarrow x \otimes y$
$[x_h, x_l] \leftarrow Split(x, q)$
$[y_h, y_l] \leftarrow Split(y, q)$
$\delta \leftarrow x_l \otimes y_l \ominus (((z \ominus x_h \otimes y_h) \ominus x_l \otimes y_h) \ominus x_h \otimes y_l)$
return (x, δ)

3.2 Approximation of the Error for /

For division "the elementary rounding error is generally not a floating point number" [11] because hardware floating point division is implemented using iterative algorithms that can accumulate error. Commonly used algorithms for division can be found in [1]. Thus, δ for division cannot be computed exactly and instead an approximation as defined in [11] is used. Given a $(\hat{x}, \hat{y}) \in \mathbb{F}^2$ pair, the approximation algorithm for division produces an output pair $(fl(\hat{x}/\hat{y}), \bar{\delta}) \in \mathbb{F}^2$ such that

$$(\hat{x}/\hat{y}) = fl(\hat{x}/\hat{y}) + \bar{\delta}$$

and $\bar{\delta}$ is bounded as

$$\left|\delta - \bar{\delta}\right| \leq \epsilon_M \left|\delta\right|$$

where δ is the true error associated with division and $\bar{\delta}$ is an approximation of δ. The idea here is that "the computed approximation is as good as" it can be expected to be "in the working precision." [11] The `ApproxTwoDiv` algorithm is given below and its Matlab

implementation can be found in B.1.

Algorithm 3.4: LANGLOIS' APPROXIMATE ALGORITHM FOR DIVISION( )

procedure TWOSUM(x, y, q)

$z \leftarrow x \oslash y$
$[v, w] \leftarrow TwoProduct(z, y, q)$
$\bar{\delta} \leftarrow (x \ominus v \ominus w) \oslash y$
return $(z, \bar{\delta})$

Notice that the `TwoSum` algorithm takes 6 flops (i.e. floating point operations), the `TwoProduct` algorithm takes 17 flops and the `ApproxTwoDiv` algorithm takes 21 flops. The count of `TwoProduct` and `ApproxTwoDiv` can be reduced to 2 and 6 flops respectively if a Fused-Multiply-and-Add (FMA) instruction is available. Thus, the elementary error calculations are not time consuming operations as they "do not use conditional branches nor attempt to access the mantissa" and instead only introduce overhead in the form of more elementary floating point instructions. [11]

4 The CENA Method

Correction Linéaire Automatique des Erreurs d'Arrondi (CENA) is a method for automatically compensating for rounding errors that result from elementary floating point operations. The method, also known as Automatic Linear Correction of Rounding Errors in English, was introduced by Philippe Langlois in 1999 and a revised version was issued in 2001. [9] The method utilizes the standard model of arithmetic along with the retrievable elementary error information to compute the global forward error of a computation and then correct the computed "naive" biased result.

4.1 The Method

Beginning with a motivating example, let $\hat{f}$ be the floating point evaluation of a real function f at the points $X = (x_1, \ldots, x_n)$ where intermediate variables $\hat{x}_{n+1}, \ldots, \hat{x}_{N-1}$ are computed to get the final results $\hat{x}_N$. [9] Then the global forward error is given by

$$\hat{x}_N - f(X) = \Delta_L - E_L$$

where "Δ_L is the first-order approximate of the global rounding error with respect to the absolute elementary rounding errors $\delta = (\delta_{n+1}, \ldots, \delta_N)$ generated by the computation of

corresponding intermediate variables." [9] Then, Δ_L is given by

$$\Delta_L = \sum_{k=n+1}^{N} \frac{\partial \hat{f}}{\partial \delta_k}(X, \delta) \cdot \delta_k$$

and E_L is the linearization error associated with the computation of Δ_L. In practice, the computation of bounds for the global error can be too inaccurate or too large to be of any feasible use. Thus, Langlois presents "a new differential method where the elementary rounding errors δ_k" are not bounded but instead computed. [9] CENA computes the global forward error with respect to the elementary rounding errors such that

$$\bar{\Delta}_L = fl(\Delta_L)$$

and gives the following correction

$$\bar{x}_N = fl(\hat{x}_N - \bar{\Delta}_L).$$

As identified by Langlois, "such a linear correction is particularly suitable when the global forward error is dominated by the first-order terms" with linear algorithms such as Horner's method for polynomial evaluation, substitution algorithm for triangular systems and Newton's method for one-dimensional zero finding. In simple terms, the correction is the sum of δ_k generated by the intermediary operations $\hat{x}_k = fl(x_i \circ x_j)$ when evaluating some function $\hat{f}$. The values of δ_k are computed using the EFTs and `ApproxTwoDiv` algorithms described above and upon computing $\hat{x}_N$, the result is corrected to get the result $\bar{x}_N$. It is also possible to correct the itermediate results when evaluating $\hat{f}$ if there is suspicion that there might be either catastrophic cancellation when evaluating $\bar{x}_N$ or an overaccumlation of rounding errors due to complex calculations. Correcting in such cases will lead to more accurate results compared to not correcting at all but these corrections may have arbitrarily bad accuracies with respect to the actual function value $f(X)$. [9] Thus, choosing intermediate results to correct throughout the computation of $\bar{x}_N$ is a viable strategy.

4.2 Further Reading

Langlois also develops a methodology for implementing a automatic differentiation framework to automate the process of calculating elementary errors along with bounds related to the computation of the derivatives and certain confidence intervals. Further details of such bounds and refinements are out of the scope of this project but can be found in [9], [11] and [5] and [12].

5 Newton's Method

5.1 Motivation

Newton's method provides a linear way of approximating a zero of some function $f \in \mathbb{R}$ provided that either the first derivative of f is known or a good estimate of it can be

obtained. A single step of a Newton iterate is given by

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

and the process is carried out until either convergence or a user-specified tolerance level is reached. For most cases, the Newton iterates will converge quadratically to a zero of f as long as a "good" starting point x_0 is chosen. However, other than issues of choosing an adequate x_0 or knowing f', an important challenge faced with Newton's method is the complexity of the function being evaluated. As the complexity increases, the probability of obtaining incorrect results due to rounding error, in general, also increases. For example, consider the function

$$f(x) = (x-1)^{11}$$

which involves 11 flops for evaluation (1 for the subtraction and 10 for the power operation). If we were instead to consider the expansion of $f(x)$, say $\tilde{f}(x)$, such that

$$\tilde{f}(x) = x^{11} - 11x^{10} + 55x^9 - 165x^8 + 330x^7 - 462x^6 + 462x^5 - 330x^4 + 165x^3 - 55x^2 + 11x - 1$$

then evaluation of the function will require 77 flops. A similar analysis shows that the derivative of f will require 11 flops in factored form and 66 flops in the expanded form. Combining everything together, Newton's method on f in factored form will require 24 flops while the expanded form will require 145 flops to compute a single step in the iteration. It is easy to see that using $\tilde{f}$ over f may lead to rounding errors as either the number of iterations are increased or the precision of the computation is decreased. Yet, there are also benefits in decreasing the working precision as, in general, lowering precision leads to increased performance. Thus, the CENA method has great potential in providing increased accuracy when working in lower precision while introducing little overhead if an FMA command is available and "acceptable" overhead otherwise.

5.2 Implementation

Three different implementations of Newton's method is used for experimental evaluation. The first is a Naive implementation with no compensation of errors and the Matlab code for it is given in C.1. The second implementation still doesn't do any error compensation but computes the values of the target polynomial using Horner's method for polynomial evaluation for increased accuracy. The Matlab code for this is given in C.2. Finally, the CENA method is implemented with a compensated Horner's scheme as presented in [12] along with the use of `ApproxTwoDiv` to compensate for the division. The CENA compensated Matlab code can be found in C.3. For all three methods all valued are computed in single precision floating point arithmetic. The functions being evaluated will be the algebraic expeansions of

$$f^n(x) = (x-1)^n \quad \text{for } n = 2, \ldots, 11.$$

The choice of these functions arise from the fact that the only zero of f^n is 1 for all n but there is an expectation of increasing errors as n increases. For each f^n, two starting points of $x_0^1 = 2$ and $x_0^2 = 11/6$ is considered since the former is exactly representable and the latter is not. The combination of these functions and starting values should provide an hollistic view on the effectiveness of CENA. Additionally, for each function-starting point combination the naive method is run in double precision to test the hypothesis that "the computed result" of the CENA method "is as accurate as if it was computed" without any compensation. [12] [9] [11]

The evaluation conditions consists of running each method for 30 iterations with 0 tolerance to observe the overall behavior of the Newton iterates in each case. The choice of 30 iterations is from [10] where Langlois gives results for a CENA compensated method for f^6 with starting point of $x_0 = 2$. In fact, the corrected iterates from [10] converge exactly to 1 in 30 iterations while the uncompensated single and double precision are less accurate and fail to terminate despite running over 100 iterations. The 30 iteration cutoff will also serve to verify these results.

Overall, the iterates should monotonically converge to the zeroes of f^n for all n from either above or below x_0. Thus, one of the evaluation criteria is "deviations from expected behavior" (DFEB) which is a measure of the amount of times a run of Newton's method either diverges from monotonicity or overshoots the value of the target zero of the function. The second criterion is the ratio of the absolute error between the target zero and the smallest observed output from the Newton iterates to the absolute value of the target zero. This value is also known as the minimum relative error $re(x^i_{\min})$ which is given by

$$re(x^i_{\min}) = \frac{|x^i_* - x^i_{\min}|}{|x^i_*|} \quad \text{where } i = 2, \ldots, 11 \quad \text{and } x^i_* = 1 \text{ for all } n$$

and can be used to measure the forward stability of the CENA method.

5.3 Results

The results for f^n when started on $x_0 = 2$ is given in Table 1. Similarly, the results for f^n when started on $x_0 = 11/6$ is given in Table 2. For both tables, values in bold denote the lowest amount of relative error achieved for a row while the underlined values denote the second lowest amount of relative error achieved for a row.

	Method							
	Naive		Horner		CENA		Double	
n	DFEB	re	DFEB	re	DFEB	re	DFEB	re
2	0	2.44e-4	0	1.22e-4	0	**0**	0	**0**
3	0	4.50e-3	0	2.67e-3	4	6.90e-4	0	**5.57e-6**
4	3	5.33e-3	0	1.87e-2	1	**2.42e-5**	0	1.80e-4
5	1	1.41e-2	1	1.82e-2	5	2.03e-2	0	**1.26e-3**
6	3	5.37e-2	4	5.32e-2	2	3.57e-2	0	**4.20e-3**
7	6	8.76e-2	2	4.19e-2	3	**7.58e-3**	0	9.78e-3
8	5	1.36e-1	4	6.93e-2	3	3.97e-2	0	**1.78e-2**
9	4	5.29e-2	6	1.34e-1	5	3.59e-2	0	**2.77e-2**
10	3	2.88e-1	3	2.41e-1	7	1.01e-1	0	**4.38e-2**
11	4	1.03e-1	7	3.88e-2	3	1.47e-1	0	**5.76e-2**

Table 1: Results for f^n with $x_0 = 2$.

	Method							
	Naive		Horner		CENA		Double	
n	DFEB	re	DFEB	re	DFEB	re	DFEB	re
2	0	2.11e-5	0	7.45e-5	0	**0**	0	**0**
3	0	6.47e-4	0	3.64e-3	3	3.58e-3	0	**3.47e-6**
4	4	5.36e-3	2	9.03e-3	2	5.54e-3	0	**1.72e-4**
5	1	3.49e-2	2	1.51e-2	7	1.51e-2	0	**1.05e-3**
6	3	5.37e-2	4	5.31e-2	2	3.57e-2	0	**4.09e-3**
7	6	7.75e-3	4	2.77e-2	2	3.88e-2	0	**6.86e-3**
8	1	4.25e-2	5	3.38e-2	4	2.19e-2	0	**8.56e-3**
9	18	1.36e-1	6	3.42e-2	3	**1.10e-2**	0	1.18e-2
10	4	1.85e-1	4	2.01e-1	4	1.12e-1	0	**3.61e-3**
11	4	1.91e-1	7	1.96e-1	4	1.47e-1	1	**4.14e-2**

Table 2: Results for f^n with $x_0 = 11/6$.

5.4 Analysis

It can be clearly seen from both tables that computing the CENA corrections leads to much more accurate results compared to no correction at all. However, most of the time the CENA correction does not perform "just as well" as computing naively in double precision. For Table 1 there is an outlier at $n = 5$ where the single precision naive Newton performs the second best and $n = 11$ where the single precision uncompensated Horner's method performs the second best. Similarly, outliers for Table 2 occur at $n = 5$ and $n = 7$ where the Horner and naive methods both perform the second best, respectively. There is also an outlier with the DFEB measurement for naive Newton at $n = 9$ for Table 2. This is because the iterates started producing `NaN` values after a few iterations which meant that each invalid iteration had to be counted as a deviation from expected behavior.

When looking at the DFEB measurement, no strong correlation between DFEB and the relative error can be observed. In fact, all single precision methods display large deviations from the monotonic expected behavior for different situations. This in turn makes it difficult to predict how the lowest relative error would have turned if the experiment, for example, were run for 50 iterations instead of the 30. Overall, all methods when run for 30 iterations do approach the zero with varying levels of success and rounding instability. The DFEB speaks mostly to how hard it is to choose a stopping criteria for an unstable Newton's method as letting the method run longer may either lead to more accurate or inaccurate results. Furthermore, it could also lead to either divergence or a total failure due to catastrophic rounding errors. The CENA method, from observations made when carrying out the experiments, is usually more stable in the sense that the other two single precision methods will either fail or diverge after around 100 iterations while the CENA method can run for longer amounts and still converge to the zero from both sides.

While the compensated Newton's method is better than naive implementations, it is certainly not more effective than doubling the working precision in the case for f^n for almost all n. Furthermore, for f^6 it does not converge exactly to $x^* = 1$ when starting with $x_0 = 2$ as shown in [10]. This could be due to possible errors introduced in the implementation of the correction method. However, assuming that there are no implementation issues then it is easy to see that CENA compensated Newton's method does not outperform doubling the precision. Additionally, no runtime analysis has been performed to measure the computational costs incurred for using compensation in single precision compared to just increasing the precision to double. While it is claimed that in [10] that the CENA method introduces minimal overhead for solving triangular systems using back substitution and Horner's method for polynomial evaluation, this analysis should be verified for Newton's method to get a truer understanding of the benefits of using the corrected Newton's method.

Finally, looking at the DFEB values allows for the criticism that the more accurate results could just be a product of "luck." Arbritrarily changing the number of iterations the methods run for can lead to arbitrarily more or less accurate relative errors. While it is reasonable to use 30 iterations since anything beyond it defeats the quadratic convergence gains expected from Newton's method, a more robust analysis could also take into the account the magnitude of these deviations from the monotonic behavior. Yet, from casual observation it can be said

that the CENA compensations lead to a more stable algorithm.

Overall, the CENA error compensating Newton's method is an improvement to naive and Horner evaluated Newton's methods when working in single precision. It is reasonable to expect that a CENA Newton's method when working in double precision should outperform its double precision naive counterpart. However, this has not been verified. Furthermore, using the CENA method is, in general, not as "good" as doubling the working precision of naive implementations.

6 Evaluation

6.1 Flaws in Analysis

The greatest issue for the experiments is that Matlab does not allow for the user to change optimization levels or specifiy a global working precision. The default precision is double and if an elementary operation occurs such that one operand is single and the other is double, the former is first converted to double precision and then the result is converted back into single precision. This has the obvious issue of not being the same operation as having both operands be in single precision. Thus, a preprocessing step of casting all initial values such as the starting point and the coefficients of the function being evaluated to single precision has been done. The assumption here is that if all operands are initialized as single then a program should work as single and Matlab will take care of the rest of the type casts.

Yet, this does not solve the issue completely since there is no way to have control over the optimizations the Matlab interpreter may make. For example, Matlab may introduce temporary variables that are in double precision under the hood or decide to use the extended precision registers available with x86 architecture machines (and the machine used for the experiments falls under this category). Another issue is that when evaluating the EFTs such as the `TwoSum` algorithm given in A.1 a smart interpreter/compiler may realize that the line `left-(val-s)` is equivalent to `left-(val-val+left)` which is then equivalent to `0` and consequently never execute this instruction. However, such relations that hold true over infinite precision are not generally correct for floating point computations and thus the optimization should not be made. It appears that Matlab is not doing this but this has not been verified. Finer points regarding issues when verifiying floating points computations can be found in [16].

Criticisms regarding the experimental evaluation have already been given in Section 5.4. Further work could involve considering more types of functions (especially polynomials), run-time analysis to obtain a benefit-to-overhead understanding and an exploration of other bounds Langlois introduces in [9] to better understand how well the correction being made is.

6.2 Evaluation of CENA

While not discussed in this project, the error analysis of the CENA method (or methods)

is tightly intertwined with the conditioning of a problem. [9] [10] This is naturally the case since errors are computed rather than formally approximated or estimated in some other way. For example, consider another work by Langlois which deals with solving triangular systems using the substitution algorithm. [11] The paper considers systems of size $n \times n$ which become very ill-conditioned as n increases. The relative forward error in the corrected single-precision solution also increases as the conditioning gets worse. This means that the effectiveness of the correction decreases as the need for sound corrections increases. This is also applicable to the corrected Newton's method algorithm since as the degree of the polynomial increases the accuracy of the corrected result decreases. Overall, it seems that the greatest advantage of CENA corrections is also its greatest weakness since errors are not approximated or reasoned about but instead computed. Thus, the CENA corrected Newton's method appears to be useful as possibly an optional parameter that could be activated for numerical software implementing Newton's method where a warning indicating ill-conditioning would also be necessary. Perhaps a more exciting direction would be that it could be used as an educational tool to demonstrate issues concering rounding error and the ideas behind methods that attempt to compensate forward error. Yet, CENA corrections should not be used for critical applications where the accuracy of a result is vital and instead other options including doubling the working precision should be considered.

7 Conclusion

The CENA method computes forward error in linear algorithms and makes corrections to offset for rounding error introduced by elementary operations in finite precision arithmetic. This paper presents the application of the CENA compensations on Newton's method for one-dimensional zero finding of functions of increasing degree but same root. Experimental results show that while the CENA leads to lower relative error in general, it is not equivalent to doubling the working precision. Thus, the study shows that the CENA method may not be suitable for critical numerical applications while further testing may be needed as outlined. Instead, a different direction for the CENA method is identified as it shows promise for highlighting issues related to rounding errors and compensatory frameworks related to such errors.

Appendices

A Matlab Code for EFTs

A.1 TwoSum

```
function [val, err] = two_sum(left, right)
% Error free transformation for addition/subtraction
% (OP) of two floating points numbers of any
% (but same) precision.
% Inputs:   left -- a floating point number such that
%                   left = fl(left)
%          right -- a floating point number such that
%                   right = fl(right).
% Let the error in fl(leftOPright) be expressed as:
% error = (leftOPright)-fl(leftOPright)
% Then the left of sum allows us to compute:
% (leftOPright) = fl(leftOPright) + error.
% Outputs:   val -- fl(leftOPright)
%            err -- error as defined above (i.e.
%                   elementary rounding error).
val = left+right;
s = val-left;
err = (left-(val-s))+(right-s);
end
```

A.2 Split

```
function [p1, p2] = split_high_low(val, q)
% Split function that splits a floating point number
% to two parts, such that p1+p2=val with each part
% having at most r-1 non-zero bits, r is defined as
% the floor of q (i.e. the precision of val).
% Inputs:   val -- sum number val = fl(val)
%             q -- the # bits of val's mantissa.
% Outputs:   p1 -- the first part of val
%            p2 -- the second part of val.
r = ceil(q/2);
inter = val*((2^r)+1);
p1 = inter-(inter-val);
p2 = val-p1;
end
```

A.3 TwoProduct

```
function [val, err] = two_product(left, right, q)
% Error free transformation for multiplication (OP)
% of two floating points numbers of any (but same)
% precision.
% Inputs:   left -- a floating points number such that
%                   left = fl(left)
%          right -- a floating point number such that
%                   right = fl(right)
%              q -- the # bits of left and rights's
%                   mantissas this implies both left
%                   and right must be of the same
%                   precision.
% Outputs:   val -- fl(leftOPright)
%            err -- error as defined above (i.e.
%                   elementary rounding error).
val = left*right;
[left_h, left_l] = split_high_low(left,q);
[right_h, right_l] = split_high_low(right,q);
err = left_l*right_l-(((val-left_h*right_h)-left_l*right_h)...
      -left_h*right_l);
end
```

B Matlab Code for Division Approximation

B.1 Approximate Division

```
function [val, err] = approx_two_div(left, right, q)
% NON-error free transformation for division (OP)
% (since an error free transformation is not possible)
% of two floating points numbers of any (but same)
% precision.
% Inputs:   left -- a floating points number such that
%                   left = fl(left)
%          right -- a floating point number such that
%                   right = fl(right)
%              q -- the # bits of left and rights's
%                   mantissas this implies both left
%                   and right must be of the same
%                   precision.
% Outputs:   val -- fl(leftOPright)
%            err -- error as defined above (i.e.
```

```
%                       approx. division rounding error).
assert(right~=0,'A division by zero has occurred!')
val = left/right;
[temp_val, temp_err] = two_product(val,right,q);
err = (temp_val-left-temp_err)/right;
end
```

C Matlab Code for Newton's Method

C.1 Naive Newton's Method

```
function zero = newton_naive(fx,fdx,x0,maxit,ftol,target)
% Naive implementation of newton's method.
% Performs maxit iterations of newton's method on function
% f as defined below and uses function fdx to compute the
% derivative of f at x.
% Inputs:      fx -- anonymous function to find the zero of
%             fdx -- derivative of f (also anonymous)
%              x0 -- initial starting point, defined by user
%           maxit -- the maximum number of iterations for the
   algorithm
%            ftol -- user specified tolerance
%          target -- known zero (we wish to find) for error
   analysis.
% Outputs:   zero -- the estimated solution of the zero of f.
err = abs(x0-target);
sprintf('%.7e | %.7e | %.7e | %.7e | %.7e',0,x0,fx(x0),fdx(x0),err
   )
for k=1:maxit
    x0 = x0 - fx(x0)/fdx(x0);
    err = abs(x0-target);
    sprintf('%.7e | %.7e | %.7e | %.7e | %.7e',k,x0,fx(x0),fdx(x0)
       ,err)
    if fx(x0) == 0
        disp('Found f(x) = 0!')
        break
    elseif abs(fx(x0)) < ftol
        disp('|f| is within tolerance!')
        break
    end
end
zero=x0;
```

```
28 end
```

C.2 Horner's Evaluated Newton's Method

```
1  function zero = newton_horner(f,x0,maxit,ftol,target)
2  % Horner's method augmented implementation of newton's method.
3  % Performs maxit iterations of newton's method on function
4  % f as defined below and uses function fdx to compute the
5  % derivative of f at x. The function evaluations are done
6  % using an uncompensated Horner's algorithm.
7  % Inputs:      fx -- symbolic function to find the zero of
8  %              x0 -- initial starting point, defined by user
9  %           maxit -- the maximum number of iterations for the
       algorithm
10 %            ftol -- user specified tolerance
11 %          target -- known zero (we wish to find) for error
       analysis.
12 % Outputs:   zero -- the estimated solution of the zero of f.
13 f_x = expand(f);
14 f_dx = diff(f_x);
15 coef_x = sym2poly(f_x);
16 coef_dx = sym2poly(f_dx);
17 coef_x = single(coef_x);
18 coef_dx = single(coef_dx);
19 err = abs(x0-target);
20 sprintf('%.7e | %.7e | %.7e | %.7e | %.7e',0,x0,horner(coef_x,x0)
       ,...
21     horner(coef_dx,x0),err)
22 for k=1:maxit
23     f_x_res = horner(coef_x,x0);
24     f_dx_res = horner(coef_dx, x0);
25     x0 = x0 - f_x_res/f_dx_res;
26     err = abs(x0-target);
27     sprintf('%.7e | %.7e | %.7e | %.7e | %.7e',k,x0,f_x_res,
           f_dx_res,err)
28     if f_x_res == 0
29         disp('Found f(x) = 0!')
30         break
31     elseif abs(f_x_res) < ftol
32         disp('|f| is within tolerance!')
33         break
34     end
35 end
```

```
zero=x0;
end

function val = horner(vals,x0)
val = vals(1);
for i=2:length(vals)
    val = (val*x0)+vals(i);
end
end
```

C.3 CENA Compensated Newton's Method

```
function zero = newton_cena(f,x0,maxit,ftol,target,q)
% CENA implementation of newton's method
% Performs maxit iterations of compensated newton's method
% on function f as defined below and compuets the derivative
% of x using Matlab's built in auto-differentiation package
% Inputs:       f -- symbolic function to find the zero of
%              x0 -- initial starting point, defined by user
%           maxit -- the maximum number of iterations for the
   algorithm
%            ftol -- user specified tolerance
%               q -- the # bits of the used precision's mantissa
%          target -- known zero (we wish to find) for error
   analysis.
% Outputs:   zero -- the estimated solution of the zero of f.
f_x = expand(f);
f_dx = diff(f_x);
coef_x = sym2poly(f_x);
coef_dx = sym2poly(f_dx);
coef_x = single(coef_x);
coef_dx = single(coef_dx);
for k=1:maxit
    [f_x_val, f_x_er] = comp_horner(coef_x, x0, q);
    [f_dx_val, f_dx_er] = comp_horner(coef_dx, x0, q);
    f_x_res = f_x_val+f_x_er;
    f_dx_res = f_dx_val+f_dx_er;
    [div, div_err] = approx_two_div(f_x_res, f_dx_res, q);
    div = div+div_err;
    [res, res_err] = two_sum(x0, -1*div);
    err = abs(x0-target);
    sprintf('%.7e | %.7e | %.7e | %.7e | %.7e',k,x0,f_x_res,
       f_dx_res,err)
```

```
29      x0 = res+res_err;
30      if f_x_res == 0
31          disp('Found f(x) = 0!')
32          break
33      elseif abs(f_x_res) < ftol
34          disp('|f| is within tolerance!')
35          break
36      end
37  end
38  zero = x0;
39  end
40
41  function [val, err] = comp_horner(vals, x, q)
42  val = vals(1);
43  pi = eye(1, length(vals)-1);
44  sigma = eye(1, length(vals)-1);
45  for i=2:length(vals)
46      [mul_val, mul_err] = two_product(val, x, q);
47      [sum_val, sum_err] = two_sum(vals(i), mul_val);
48      val = sum_val;
49      pi(i) = mul_err;
50      sigma(i) = sum_err;
51  end
52  assert(length(pi)==length(sigma),'Issue at compensated Horner')
53  err = pi(length(pi))+sigma(length(sigma));
54  for i=2:length(pi)
55      err = (pi(i)+sigma(i))+(err*x);
56  end
57  end
```

References

[1] Parhami Behrooz. Computer arithmetic: Algorithms and hardware designs. *Oxford University Press*, 19:512583–512585, 2000.

[2] Sylvie Boldo, Jacques-Henri Jourdan, Xavier Leroy, and Guillaume Melquiond. Verified compilation of floating-point computations. *Journal of Automated Reasoning*, 54(2):135–163, 2015.

[3] Eva Darulova and Viktor Kuncak. Sound compilation of reals. In *Acm Sigplan Notices*, volume 49, pages 235–248. ACM, 2014.

[4] Theodorus Jozef Dekker. A floating-point technique for extending the available precision. *Numerische Mathematik*, 18(3):224–242, 1971.

[5] Stef Graillat, Philippe Langlois, and Nicolas Louvet. Algorithms for accurate, validated and fast polynomial evaluation. *Japan Journal of Industrial and Applied Mathematics*, 26(2-3):191–214, 2009.

[6] Nicholas J Higham. *Accuracy and stability of numerical algorithms*, volume 80. Siam, 2002.

[7] Donald E Knuth. The art of computer programming, 2: seminumerical algorithms, addision wesley. *Reading, MA*, 1998.

[8] Michael O Lam and Jeffrey K Hollingsworth. Fine-grained floating-point precision analysis. *The International Journal of High Performance Computing Applications*, page 1094342016652462, 2016.

[9] Philippe Langlois. Automatic linear correction of rounding errors. *BIT Numerical Mathematics*, 41(3):515–539, 2001.

[10] Philippe Langlois. More accuracy at fixed precision. *Journal of computational and applied mathematics*, 162(1):57–77, 2004.

[11] Philippe Langlois and Nicolas Louvet. Solving triangular systems more accurately and efficiently. In *Proceedings of the 17th IMACS World Congress, Paris*, volume 400, pages 1–10, 2005.

[12] Philippe Langlois and Nicolas Louvet. How to ensure a faithful polynomial evaluation with the compensated horner algorithm. In *Computer Arithmetic, 2007. ARITH'07. 18th IEEE Symposium on*, pages 141–149. IEEE, 2007.

[13] Jacques-Louis Lions et al. Ariane 5 flight 501 failure, 1996.

[14] James Mackintosh. Beware lessons of history when dealing with quirky indices. *Financial Times*, Aug 2015.

[15] Matthieu Martel. Floating-point format inference in mixed-precision. In *NASA Formal Methods Symposium*, pages 230–246. Springer, 2017.

[16] David Monniaux. The pitfalls of verifying floating-point computations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(3):12, 2008.

[17] Takeshi Ogita, Siegfried M Rump, and Shin'ichi Oishi. Accurate sum and dot product. *SIAM Journal on Scientific Computing*, 26(6):1955–1988, 2005.

[18] Cindy Rubio-González, Cuong Nguyen, Benjamin Mehne, Koushik Sen, James Demmel, William Kahan, Costin Iancu, Wim Lavrijsen, David H Bailey, and David Hough. Floating-point precision tuning using blame analysis. In *Proceedings of the 38th International Conference on Software Engineering*, pages 1074–1085. ACM, 2016.

[19] HP Sharangpani and ML Barton. Statistical analysis of floating point flaw in the pentiumtm processor (1994). *Intel Corporation*, 30, 1994.

[20] Robert Skeel. Roundoff error and the patriot missile.

[21] Laura Titolo, Marco A Feliú, Mariano Moscato, and César A Munoz. An abstract interpretation framework for the round-off error analysis of floating-point programs. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 516–537. Springer, 2018.

[22] Dan Zuras, Mike Cowlishaw, Alex Aiken, Matthew Applegate, David Bailey, Steve Bass, Dileep Bhandarkar, Mahesh Bhat, David Bindel, Sylvie Boldo, et al. Ieee standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–70, 2008.