Fast, robust, and accurate low-rank LDL^T quartic equation solver

Wilco Oelen

May 1, 2023

Abstract

A quartic expression $Q(z) = z^4 + Az^3 + Bz^2 + Cz + D$ with real coefficients A, B, C, D can be written as a quadratic form $Q(z) = \mathbf{z}^{\mathrm{T}}\mathbf{Q}(\varphi)\mathbf{z}$, where $\mathbf{z} = [z^2, z, 1]^{\mathrm{T}}$ and $\mathbf{Q}(\varphi)$ is a 3×3 real symmetric matrix, depending on a parameter φ, which only appears in the antidiagonal elements of the 3×3 matrix. The parameter φ can be chosen, such that the rank of $\mathbf{Q}(\varphi)$ drops to 2. The algorithm, presented in this paper, is based on the above observation, as exploited by P. Strobach [5], but it avoids the introduction of severe cancellation errors and takes care of exceptional cases where Strobach's algorithm fails. A Java program, which implements the algorithm, combined with an extensive test suite, demonstrates the exceptional performance of the algorithm, in terms of speed, accuracy, and robustness. The performance of the code is compared to the performance of other well-established solvers. Up to now, not a single quartic equation could be found which cannot be handled satisfactorily by the new algorithm.

1 Introduction

Quartic equations can be solved analytically. An analytic solution of the general monic quartic equation with real coefficients, $z^4 + Az^3 + Bz^2 + Cz + D = 0$, dates back to the 16th century. It was discovered by Lodovico Ferrari [1]. Using this solution in a software implementation leads to numerical problems for many quartic equations, due to cancellation errors [2]. Recently, alternative analytic solutions have been proposed and methods have been suggested to reduce the numerical problems. However, all of these solutions and methods suffer from numerical cancellation errors in specific configurations of the coefficients of the given quartic equation, [3], [4], [5], [6]. A different approach is to use purely numerical methods. There are several well-established general purpose polynomial equation solvers, e.g. [7], [8], [9], and there also are numerical methods, specifically designed for quartic equations [2]. Some of the best numerical methods give accurate results, combined with good efficiency (MPSOLVE, based on [9], and especially PA17, based on [8]), but even these efficient methods still are several times more computationally intensive than well-crafted implementations of one of the analytic solutions.

In this paper, an algorithm is presented, which is accurate and robust, and at the same time efficient in terms of computational speed. It only is a few tens of percents slower than a well-crafted purely analytical algorithm, running on the same hardware and using the same computer language and compiler settings.

In chapter 2 an outline of Strobach's algorithm, see [5], is presented, and it is shown how Strobach handles numerical cancellation. In chapter 3 a new algorithm, largely based on Strobach's algorithm, is presented. It is explained where Strobach's algorithm fails and how the new algorithm solves these issues. In chapter 4 a description is given of the software implementation and how it is compared to other methods. In chapter 5, results are presented, both in terms of accuracy, and in terms of speed.

2 An outline of the quartic solver algorithm

All analytic solution methods up to very recently (before Strobach's algorithm, presented in 2015) for the quartic equation with real coefficients, first perform a substitution $z = y - \frac{A}{4}$ to get an equation in y, which is of the form $y^4 + py^2 + qy + r = 0$, where p, q and r are real numbers, which can be expressed as polynomials in A, B, C, and D. This is the so-called *depressed form* of the quartic equation, which has no third-degree term. Next, the solution methods attempt to write the depressed quartic form as a product of two quadratic factors, with real coefficients s, t, u, and v: $y^4 + py^2 + qy + r = (y^2 + sy + t)(y^2 + uy + v)$. Different methods were discovered to obtain values for s, t, u, and v by Ferrari [1], Descartes [10] and Euler [11], but all these methods eventually boil down to solving a *cubic resolvent*, which is the same for all these methods. Numerical cancellation errors can occur in multiple steps of the solutions. Software implementations of these methods produce inferior results for tiny roots in cases where the dynamic range of the roots is large. Here, the dynamic range is defined as the ratio $|z_{\max}|/|z_{\min}|$ of the largest-magnitude root $z_{\max}$ and the smallest-magnitude root $z_{\min}$.

Strobach found a new analytic solution of the quartic equation, which also leads to a cubic resolvent, but this cubic resolvent is different from the one, present in the above-mentioned methods. A very important difference is that in Strobach's method, the cubic resolvent can be computed without appreciable numerical cancellation errors and that it appears in depressed form (without quadratic term), which also allows its solution to be derived without risk of excessive loss of precision. Here, an outline is given of Strobach's algorithm. A full and detailed description can be found in his paper [5].

2.1 The quartic function as a parameterized quadratic form

Strobach's solution starts with the monic equation

$$Q(z) = z^4 + Az^3 + Bz^2 + Cz + D = 0 \tag{1}$$

where A, B, C, and D are real numbers and $D \neq 0$ (when D equals 0, one can use a 3^{th} degree equation solver).

It is easy to show that the fourth degree polynomial $Q(z)$ can be written as a quadratic form

$$Q(z) = \begin{bmatrix} z^2 & z & 1 \end{bmatrix} \mathbf{Q}(\varphi) \begin{bmatrix} z^2 \\ z \\ 1 \end{bmatrix} \tag{2}$$

where

$$\mathbf{Q}(\varphi) = \begin{bmatrix} 1 & \frac{A}{2} & (\frac{B}{6} + \frac{\varphi}{2}) \\ \frac{A}{2} & (\frac{2B}{3} - \varphi) & \frac{C}{2} \\ (\frac{B}{6} + \frac{\varphi}{2}) & \frac{C}{2} & D \end{bmatrix} \tag{3}$$

The φ cancels out perfectly, when this expression is evaluated and written as a scalar polynomial value in z. This property gives us a parameter, which can be freely chosen, without affecting $Q(z)$.

2.2 Reducing the rank of the quadratic form matrix

In general, the rank of the matrix $\mathbf{Q}(\varphi)$ equals 3, but it is possible to choose φ, such that the rank of this matrix drops to 2. This is the case if $det(\mathbf{Q}(\varphi)) = 0$. The determinant of this matrix can be written as a *depressed* cubic:

$$det(\mathbf{Q}(\varphi) = \varphi^3 + g\varphi + h \tag{4}$$

with coefficients g and h, where

$$g = AC - 4D - \frac{B^2}{3} \quad ; \quad h = \left(8D + AC - \frac{2B^2}{9}\right)\frac{B}{3} - C^2 - DA^2 \quad . \tag{5}$$

Strobach developed an algorithm for evaluation of g and h from the coefficients A, B, C, and D, which does not suffer from numerical cancellation. He also developed a simple and fast numerical algorithm for finding the *dominant* root φ_0 of the depressed cubic. The dominant root is the root which is well-conditioned and which can be determined with high accuracy for all g and h. For real g and h, this dominant root is real as well. For here, it is sufficient to assume that the depressed cubic (4) can be determined accurately and that a real root φ_0 can be found accurately (up to machine precision accuracy) with high efficiency. All details are given in Strobach's paper [5], chapter 3.

The matrix $\mathbf{Q}(\varphi_0)$ has rank at most equal to 2 and this allows it to be decomposed into rank-2 $\mathrm{LDL}^{\mathrm{T}}$ factors:

$$\mathbf{Q}(\varphi_0) = \begin{bmatrix} 1 & 0 \\ \ell_1 & 1 \\ \ell_3 & \ell_2 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & d_2 \end{bmatrix} \begin{bmatrix} 1 & \ell_1 & \ell_3 \\ 0 & 1 & \ell_2 \end{bmatrix} \tag{6}$$

The elements ℓ_1, ℓ_2, ℓ_3, and d_2 can be derived analytically. How this is done is covered in chapter 3 of this paper.

2.3 Factorization into quadratics, using rank-2 decomposition

Once we have the elements ℓ_1, ℓ_2, ℓ_3, and d_2, as described in equation (6), it is easy to construct the two quadratic factors of the given quartic $Q(z)$. The quadratic factors in

Strobach's algorithm may be complex (each other's conjugate), as opposed to the above-mentioned methods of Ferrari, Descartes and Euler. The inner 2×2 matrix of equation (6) can be written as

$$\begin{bmatrix} 1 & 0 \\ 0 & d_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & \gamma \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & \sigma \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & \gamma \end{bmatrix} \tag{7}$$

where $\sigma = sign\{d_2\}$ (for non-zero d_2 this is either -1 or 1) and $\gamma = \sqrt{|d_2|}$.
The quartic now can be expressed as

$$Q(z) = \underbrace{\begin{bmatrix} z^2 & z & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ \ell_1 & 1 \\ \ell_3 & \ell_2 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & \gamma \end{bmatrix}}_{= \begin{bmatrix} p_1(z) & p_2(z) \end{bmatrix}} \begin{bmatrix} 1 & 0 \\ 0 & \sigma \end{bmatrix} \underbrace{\begin{bmatrix} 1 & 0 \\ 0 & \gamma \end{bmatrix} \begin{bmatrix} 1 & \ell_1 & \ell_3 \\ 0 & 1 & \ell_2 \end{bmatrix} \begin{bmatrix} z^2 \\ z \\ 1 \end{bmatrix}}_{= \begin{bmatrix} p_1(z) \\ p_2(z) \end{bmatrix}} \tag{8}$$

where $p_1(z) = z^2 + \ell_1 z + \ell_3$ and $p_2(z) = \gamma z + \gamma \ell_2$.

Now two cases can be distinguised, one for $\sigma = -1$ and one for $\sigma = 1$:

Case 1 ($\sigma = -1$, corresponding to $d_2 < 0$):

$$\begin{aligned} Q(z) &= {p_1}^2(z) - {p_2}^2(z) = (p_1 + p_2)(p_1 - p_2) \qquad \Rightarrow \\ Q(z) &= (z^2 + (\ell_1 + \gamma)z + (\ell_3 + \gamma\ell_2))(z^2 + (\ell_1 - \gamma)z + (\ell_3 - \gamma\ell_2)) \end{aligned} \tag{9}$$

Case 2 ($\sigma = 1$, corresponding to $d_2 > 0$):

$$\begin{aligned} Q(z) &= {p_1}^2(z) + {p_2}^2(z) = (p_1 + ip_2)(p_1 - ip_2) \qquad \Rightarrow \\ Q(z) &= (z^2 + (\ell_1 + i\gamma)z + (\ell_3 + i\gamma\ell_2))(z^2 + (\ell_1 - i\gamma)z + (\ell_3 - i\gamma\ell_2)) \end{aligned} \tag{10}$$

This effectively solves the quartic equation. The quartic is factored either into two real quadratic factors, or into two complex quadratic factors, which are conjugate to each other. The zeros of the quadratic factors are easily determined.

In chapter 4 of Strobach's paper, an algorithm is given for determining the elements ℓ_1, ℓ_2, ℓ_3, and d_2 without introducing excessive numerical cancellation errors. However, this algorithm is flawed and in practice, it is easy to find quartic equations for which his algorithm fails. In these cases, the value of φ_0 can be determined perfectly fine at high accuracy, but determination of ℓ_1, ℓ_2, ℓ_3, and d_2 fails either completely, or excessive cancellation errors are introduced.

The solution also does not work when d_2 equals 0. A software implementation also becomes problematic when d_2 is close to 0. In the next chapter this situation is dealt with in such a way that there is no need to exactly specify what it means for d_2 to be "close to 0". This situation is avoided completely by switching to another type of computation if needed.

3 An improved algorithm for factoring the quartic

Combining equations (3) and (6) leads to

$$\begin{bmatrix} 1 & \frac{A}{2} & (\frac{B}{6}+\frac{\varphi_0}{2}) \\ \frac{A}{2} & (\frac{2B}{3}-\varphi_0) & \frac{C}{2} \\ (\frac{B}{6}+\frac{\varphi_0}{2}) & \frac{C}{2} & D \end{bmatrix} = \begin{bmatrix} 1 & \ell_1 & \ell_3 \\ \ell_1 & {\ell_1}^2 & \ell_1\ell_3 \\ \ell_3 & \ell_1\ell_3 & {\ell_3}^2 \end{bmatrix} + d_2 \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & \ell_2 \\ 0 & \ell_2 & {\ell_2}^2 \end{bmatrix} \tag{11}$$

From this, one easily obtains five equations for the four unknowns ℓ_1, ℓ_2, ℓ_3, and d_2.

$$\ell_1 = \frac{A}{2} \tag{12}$$

$$\ell_3 = \frac{B}{6} + \frac{\varphi_0}{2} \tag{13}$$

$$d_2 = B - 2\ell_3 - {\ell_1}^2 \tag{14}$$

$$d_2\ell_2 = \frac{C}{2} - \ell_1\ell_3 \tag{15}$$

$$d_2{\ell_2}^2 = D - {\ell_3}^2 \tag{16}$$

The four unknowns can be expressed as functions of the known values A, B, C, D, and φ_0, by only using four of these five equations. So, we have one more equation than unknowns and the system seems to be overdetermined. This set of equations, however, is exactly consistent and has a unique solution. When these equations are used in a computer program, however, it may become *numerically* inconsistent, due to rounding errors and cancellation errors. The result of a naively programmed solution, based on 4 of these equations, does not need to satisfy the fifth equation, not even approximately! So, solving this system of five equations must be done very carefully.

In chapter 4 of his paper, Strobach takes the solution for ℓ_1 and ℓ_3 for granted, based on equations (12) and (13). He then reduces the problem to solving for ℓ_2 and d_2, using equations (14), (15), and (16). He also needs to handle the special case of so-called bi-quadratic equations (those are quartic equations with $A = 0$ and $C = 0$).

Strobach's solution, however, has two major flaws. The first flaw is that Strobach always attempts to compute ℓ_2 and d_2. However, there are quartics, for which it is impossible to compute these accurately. For these quartics, d_2 is close to 0, while ℓ_2 is very large. For such quartics, both quantities then cannot be computed accurately. The second flaw is that it is not safe to use equation (13) for computing ℓ_3 in all cases. If both terms have nearly equal magnitude, but opposite sign, then in the computation of ℓ_3 there is excessive loss of

accuracy, due to cancellation errors. The resulting factorization becomes inaccurate, while the polynomial is well-conditioned and there really is no need to have high loss of accuracy.

Below, it is explained how these situations have to be dealt with in order to obtain good accuracy. The first issue also is recognized by Orellana and De Michele [12] and they deal with it in their solution, but they also do not recognize the second flaw. The algorithm in this paper deals with both issues. The first issue is handled in a different way than it is done by Orellana and De Michele.

3.1 A measure of accuracy in limited precision expressions

Before it is possible to describe the algorithm, for dealing with the above-mentioned issues, it first is necessary to define a quantitative measure of the accuracy of a numerical evaluation of a mathematical expression in a CPU with limited floating point precision. Cancellation errors can only occur with addition and subtraction. Multiplication and division are not exact, but the relative error is small (for n-bit precision, the relative error is less than 2^{-n}). For each *variable* an *accuracy measure* can be defined. Here in this paper, a *variable* can be regarded as a storage unit in computer memory, in which a single numerical value with limited precision is stored. For many modern CPU architectures, the precision is 53 bits. In practice this means that almost 16 digits of accuracy can be maintained and computations are done with better than 15-digit accuracy, but less than 16-digit accuracy.

The non-negative *accuracy measure* is defined as follows: If the lower k bits of the number are uncertain, then the variable has accuracy measure equal to 2^{-k}. In a CPU architecture where numbers are stored with n bits of precision, an accuracy measure of less than 2^{-n} means that the number is totally meaningless, it lost all bits of precision. For such numbers the accuracy measure can be set to 0. For example, if for a number with 53-bit precision the accuracy measure equals 10^{-6}, this means that it only has somewhere between 9 and 10 meaningful digits (a fully accurate 53-bit number has between 15 and 16 meaningful digits and 6 of those digits are lost). If a number has an accuracy measure equal to 0.5, then one bit of precision is lost, if the number has a precision measure equal to 0.125, then 3 bits of precision are lost (which is the loss of almost one decimal digit).

The accuracy measure can be computed easily for numerical expressions. Below, the rules are given:

- The combined result Π of multiplications or divisions of m factors f_1, f_2, . . . f_m with accuracy measures a_f1, a_f2, . . . a_fm has an accuracy measure a_Π equal to $min(a_\mathrm{f1}, a_\mathrm{f2}, ... a_\mathrm{fm})$.

- The combined result Σ of additions or subtractions of m terms t_1, t_2, . . . t_m with accuracy measures a_t1, a_t2, . . . a_tm has an accuracy measure a_Σ equal to

$$\frac{|\Sigma|}{\xi + \frac{|t_1|}{a_\mathrm{t1}} + \frac{|t_2|}{a_\mathrm{t2}} + \quad . \quad . \quad . \quad + \frac{|t_\mathrm{m}|}{a_\mathrm{tm}}} \quad ,$$

  where ξ is a very small number to avoid division-by-zero errors if all terms are equal

to 0. For the common ARM, x86, and Intel x86-64 architectures, a suitable value for ξ is the smallest positive normal IEEE number, DBL_MIN, which equals 2^{-1023}.

- The accuracy measure of a real power of a number (e.g. a square root, a square, or a cube) is equal to the accuracy measure of the number itself.

These rules allows one to have estimates of the accuracy of multistep computations with arithmetic operations and input data of limited precision, and these rules do not depend on the properties of a particular CPU architecture.

3.2 Computation of γ and $\gamma\ell_2$ of section 2.3

According to equations (9) and (10), the values of ℓ_1, ℓ_3, γ, and $\gamma\ell_2$ are needed, where γ is $\sqrt{|d_2|}$. Two different strategies can be used for that.

One strategy is to compute ℓ_1, ℓ_3, ℓ_2, and d_2 explicitly and then computing γ and the coefficients. This is the strategy, used by Strobach. Strobachs algorithm gets into trouble, when d_2 is close to 0. In that case, there is excessive numerical cancellation error in the calculated value of d_2 and when equation (15) is used for computing ℓ_2, then that value also suffers from the same cancellation error.

Another approach is to compute ℓ_1 and ℓ_3 explicity, but computing γ and $\gamma\ell_2$ directly, without the intermediate computation of ℓ_2 and d_2. If d_2 is exactly 0, then ℓ_2 is not defined (infinitely large absolute value), but $d_2\ell_2^2$ is non-zero, but finite. This follows immediately from equation (16). One then also can see immediately that $d_2\ell_2$ is 0, when d_2 equals 0. The product $\gamma\ell_2$ equals $\sqrt{|d_2|}\cdot\ell_2$. Depending on the sign of d_2, the square of this equals either $d_2\ell_2^2$ or $-d_2\ell_2^2$. So, by computing $d_2\ell_2^2$ and taking a square root, one can compute $\gamma\ell_2$ directly, without the intermediate need of d_2 and ℓ_2. Computing γ now also is easy, without needing d_2 directly. One can compute $d_2\ell_2$ and when this is divided by the computed value of $\gamma\ell_2$ one gets either $\sqrt{|d_2|}$ or $-\sqrt{|d_2|}$. Equations (15) and (16) can be used to compute $d_2\ell_2$ and $d_2\ell_2^2$.

None of the two strategies, mentioned above, works fine in all situations. It is necessary to select one of these two strategies. In order to do so, the values ℓ_1, ℓ_3, d_2, $d_2\ell_2$, and $d_2\ell_2^2$ all are computed, according to equations (12 - 16), together with the accuracy measures of these five quantities. The quantity ℓ_3 is eliminated from the accuracy measures, using equation (13). This leads to the following accuracy measures:

$$a_{\ell 1} = 1 \tag{17}$$

$$a_{\ell 3} = \frac{|\ell_3|}{\xi + \frac{|B|}{6} + \frac{|\varphi_0|}{2}} \tag{18}$$

$$a_{d2} = \frac{|d_2|}{\xi + |\frac{2B}{3}| + |\varphi_0| + {\ell_1}^2} \tag{19}$$

$$a_{d2l2} = \frac{2\,|d_2\ell_2|}{\xi + |C| + \frac{|AB|}{6} + \frac{|A\,\varphi_0|}{2}} \tag{20}$$

$$a_{d2l22} = \frac{|d_2{\ell_2}^2|}{\xi + |D| + \left(\frac{|B|}{6} + \frac{|\varphi_0|}{2}\right)^2} \tag{21}$$

If $a_{d2} \geq a_{d2l22}$, then equations (12), (13), (14), and (15) are used for computing γ and $\gamma\ell_2$ through d_2, else equations (12), (13), (16), and (15) are used for computing γ and $\gamma\ell_2$ through $d_2{\ell_2}^2$.

Using this strategy, the first flaw of Strobach's algorithm is taken care of and we do not need to handle any special cases for d_2 close to 0 or equal to 0, nor do we have to decide whether d_2 is close to 0 or not. This strategy naturally selects the optimal way of computing γ and $\gamma\ell_2$.

3.3 Computing the coefficients of the quadratic factors

Once the values of ℓ_1, ℓ_3, γ, and $\gamma\ell_2$ are determined, it is possible to determine the coefficients of the quadratic factors, as given in equations (9) and (10).

In the complex case of equation (10), there is no numerical cancellation. Computation of the coefficients of the complex coefficients is straightforward. One only needs to compute the zeros of one of the quadratic factors. The zeros of the other quadratic factor simply are the complex conjugates of the already determined zeros. Solving the quartic is simple in this case.

In the real case of equation (9), the situation is more complicated. There can be considerable numerical cancellation in the computation of the coefficients of the quadratic factors. Strobach introduces four coefficients a, b, c, and d as follows:

$$Q(z) = (z^2 + az + b)(z^2 + cy + d) \tag{22}$$

The coefficients a, b, c, and d depend on ℓ_1, ℓ_3, γ, and $\gamma\ell_2$, according to equation (9). They

can be written as

$$a = \ell_1 + \gamma \tag{23}$$
$$b = \ell_3 + \gamma\ell_2 \tag{24}$$
$$c = \ell_1 - \gamma \tag{25}$$
$$d = \ell_3 - \gamma\ell_2 \tag{26}$$

If ℓ_1 and γ have close magnitudes, then either a or c suffers from large cancellation errors. If ℓ_3 and $\gamma\ell_2$ have close magnitudes, then either b or d suffers from large cancellation errors.

Using equations (17 - 21) and the rules of section (3.1), taking into account the choice of using either equation (14) or equation (16) as described in section (3.2), one easily computes the accuracy measures a_a, a_b, a_c, and a_d of the computed values a, b, c, and d.

3.4 Pairwise improvement of coefficients of the quadratic factors

Both pairs $\{a,\ c\}$ and $\{b,\ d\}$ have a high-magnitude value and a low-magnitude value. The low-magnitude value suffers most from numerical cancellation errors. One step in the improvement of the coefficients is to recompute the low-magnitude value for both pairs from the high-magnitude value. In order to do so, equation (22) is used to get relations between a, b, c, d and A, B, C, D:

$$A = a + c \tag{27}$$
$$B = ac + b + d \tag{28}$$
$$C = bc + ad \tag{29}$$
$$D = bd \tag{30}$$

For the pair $\{b,\ d\}$, Strobach uses equation (30) as follows: if $|b| > |d|$, then replace d by D/b, else replace b by D/d. This strategy also is used here. Besides replacing the coefficient with lowest magnitude, one also has to replace the associated accuracy measure! That accuracy measure is used in further steps of the algorithm. So, if $|b| > |d|$, then one also has to replace a_d by a_b, else one also has to replace a_b by a_d. After this operation, the replaced coefficient has the same accuracy measure as the high-magnitude coefficient.

For the pair $\{a,\ c\}$, Strobach uses a more involved mechanism (see [5], section 5). Here, a different strategy is used for a and c, again based on selection of the expression with the best accuracy measure. This strategy is simpler than Strobach's strategy and tests show that it works remarkably well. The three equations (27), (28), and (29) are used.

Case 1, $|a| > |c|$: compute three possible values of c:

$$c^{(1)} = A - a, \qquad c^{(2)} = \frac{B - b - d}{a} = \frac{B - 2\ell_3}{a}, \qquad c^{(3)} = \frac{C - ad}{b}$$

and the associated accuracy measures

$$a_{\rm c}^{(1)} = \frac{|c^{(1)}|}{|A| + \frac{|a|}{a_{\rm a}}}, \quad a_{\rm c}^{(2)} = min\left(\frac{|c^{(2)}|}{\xi + |B| + \frac{|2\ell_3|}{a_{\ell 3}}},\, a_{\rm a}\right), \quad a_{\rm c}^{(3)} = min\left(\frac{|c^{(3)}|}{\xi + |C| + \frac{|ad|}{min(a_a, a_d)}},\, a_{\rm b}\right)$$

and then select the computed value for c, which has the highest accuracy measure, and replace $a_{\rm c}$ by the highest accuracy measure.

Case 2, $|a| < |c|$: compute three possible values of a:

$$a^{(1)} = A - c, \qquad a^{(2)} = \frac{B - b - d}{c} = \frac{B - 2\ell_3}{c}, \qquad a^{(3)} = \frac{C - bc}{d}$$

and the associated accuracy measures

$$a_{\rm a}^{(1)} = \frac{|a^{(1)}|}{|A| + \frac{|c|}{a_{\rm c}}}, \quad a_{\rm a}^{(2)} = min\left(\frac{|a^{(2)}|}{\xi + |B| + \frac{|2\ell_3|}{a_{\ell 3}}},\, a_{\rm c}\right), \quad a_{\rm a}^{(3)} = min\left(\frac{|a^{(3)}|}{\xi + |C| + \frac{|bc|}{min(a_b, a_c)}},\, a_{\rm d}\right)$$

and then select the computed value for a, which has the highest accuracy measure, and replace $a_{\rm a}$ by the highest accuracy measure.

3.5 Simultaneous recomputation of one pair of coefficients

After the steps, described in the previous section, there still may be issues in the computed coefficients b and d. The computed value of ℓ_3 as given in equation (13) may suffer from extreme cancellation and then the accuracy measure $a_{\ell 3}$, as given in equation (18) is close to 0. Strobach did not recognize this issue, see [5]. This issue also is not recognized by Orellana et al., see [12]. In both papers, the computed value of ℓ_3 is taken for granted.

In the real case, in the expression for b and d it may be that ℓ_3 is the dominant term, while its accuracy measure is very close to 0. This results in an overall highly inaccurate computation of b and d. Even the improvement of the previous section cannot improve the quality of these coefficients. That improvement only assures that the accuracy measures of b and d equal $max(a_b,\, a_d)$, but if the best accuracy measure still is bad, then both are bad. Only the product of b and d is accurate, due to the use of equation (30), but the individual coefficients still can be off quite a lot. In order to resolve this issue, the pair $\{b,\, d\}$ is computed from the pair $\{a,\, c\}$ if the accuracy measure of b and d is less than $min(a_a,\, a_c)$. In order to do so, equations (29) and (30) are combined to create a quadratic equation in either b, or d, the other coefficient then is computed from equation (30), without any loss of accuracy.

Initial elimination of d from equations (29) and (30) leads to the following quadratic equation, and from that solution, using equation (30), d can simply be computed:

$$c\,b^2 - Cb + aD = 0\,, \qquad d = \frac{D}{b} \tag{31}$$

Initial elimination of b from equations (29) and (30) leads to the following quadratic equation, and from that solution, using equation (30), b can simply be computed:

$$a\,d^2 - Cd + cD = 0\,, \qquad b = \frac{D}{d} \tag{32}$$

Both pairs of equations have two solutions for the pair $\{b,\, d\}$. One solution is the correct solution, the other solution is a spurious one, introduced by the fact that equation (28) is not taken into account. Only one of the two solutions satisfies equation (28).

For the pair of equations (31), the two solutions for the pair $\{b,\, d\}$ are

$$b = \frac{C \pm \sqrt{C^2 - 4acD}}{2c}, \qquad d = \frac{2cD}{C \pm \sqrt{C^2 - 4acD}} = \frac{C \mp \sqrt{C^2 - 4acD}}{2a} \tag{33}$$

For the pair of equations (32), the two solutions for the pair $\{b,\, d\}$ are

$$d = \frac{C \pm \sqrt{C^2 - 4acD}}{2a}, \qquad b = \frac{2aD}{C \pm \sqrt{C^2 - 4acD}} = \frac{C \mp \sqrt{C^2 - 4acD}}{2c} \tag{34}$$

If $C > 0$, then the solution with $+$ in front of the square root is the most accurate. If $C < 0$, then the solution with $-$ in front of the square root is the most accurate. If $C = 0$, then it does not matter. If for both pairs of equations (31) and (32) the solution is taken with the highest accuracy, then one can see from (33) and (34), that one of them is the desired solution, and the other one is the spurious solution. It is not possible in advance to tell, which one is the desired solution. So, the strategy is to compute pairs $\{b,\, d\}$ from (31) and from (32), and plugging both pairs in equation (28) and only use the pair, which satisfies (28) best. Before replacing the original pair $\{b,\, d\}$, one, however, should still check the accuracy measure of the values computed, according to the most accurate solution-pair of (33) and (34). If the accuracy is too low, then the original value of the pair $\{b,\, d\}$ should be maintained.

The total strategy for possible recomputation of $\{b,\, d\}$ is summarized below:

First compute the discriminant Δ and its accuracy measure:

$$\Delta = C^2 - 4acD\,, \qquad a_\Delta = \frac{\Delta}{\xi + C^2 + \frac{|4acD|}{min(a_a, a_c)}}$$

After this initial computation, there are two possible cases:

Case 1, $|C| \geq 0$: compute the following pairs and the numerator accuracy:

$$b^{(1)} = \frac{C + \sqrt{\Delta}}{2c}, \qquad d^{(1)} = \frac{D}{b^{(1)}}$$

$$d^{(2)} = \frac{C + \sqrt{\Delta}}{2a}, \qquad b^{(2)} = \frac{D}{d^{(2)}}$$

$$a_{num} = \frac{C + \sqrt{\Delta}}{\xi + C + \frac{\sqrt{\Delta}}{a_\Delta}}$$

Case 2, $|C| < 0$: compute the following pairs and the numerator accuracy:

$$b^{(1)} = \frac{C - \sqrt{\Delta}}{2c}, \qquad d^{(1)} = \frac{D}{b^{(1)}}$$

$$d^{(2)} = \frac{C - \sqrt{\Delta}}{2a}, \qquad b^{(2)} = \frac{D}{d^{(2)}}$$

$$a_{num} = \frac{-C + \sqrt{\Delta}}{\xi - C + \frac{\sqrt{\Delta}}{a_\Delta}}$$

If $a_{num} < min(a_b, a_d)$, then do not replace the pair $\{b, d\}$ and discard the computed pairs $\{b^{(1)}, d^{(1)}\}$ and $\{b^{(2)}, d^{(2)}\}$, else perform the next step, given below.

Compute the error $\epsilon^{(1)} = |B - ac - b^{(1)} - d^{(1)}|$ and the error $\epsilon^{(2)} = |B - ac - b^{(2)} - d^{(2)}|$. If $\epsilon^{(1)} < \epsilon^{(2)}$ then replace $\{b, d\}$ with $\{b^{(1)}, d^{(1)}\}$, otherwise replace $\{b, d\}$ with $\{b^{(2)}, d^{(2)}\}$.

Mathematically, in the above steps, the value of $\Delta = C^2 - 4acD$ is non-negative, but in a computer implementation, this value may become negative (a very small value, just below zero, due to numerical rounding noise). In a practical implementation, such a tiny negative value should be regarded as being 0.

3.6 Final step: determining the roots of the quadratic factors

The above steps assure the accurate computation of the coefficients of equation (9), and the quadratic factors now can be further factorized to get the roots of the quartic. As a final step, one could use some iterative method to polish the computed coefficients further, as is done by Orellana and De Michelle (see [12], section 2.3), but tests revealed that this hardly is necessary. Polishing also introduces the risk of iterations being sent off to far away values and special precautions need to be taken to detect such catastrophic failures. The polishing

step adds a fairly high percentage of total computational cost to the algorithm, while the added accuracy is questionable. The algorithm, presented above, reaches a *total relative accuracy* of well over 14 digits (total relative accuracy will be defined rigorously below) on an Intel x86-64 CPU with standard IEEE 53 bits arithmetic, without any polishing.

So, the coefficients of the quadratic factors are used as is, and the problem of solving the quartic now is reduced to the problem of solving two monic quadratic equations. The solution of the quadratic equation $z^2 + az + b = 0$ can b written as $z_{1,2} = -\frac{1}{2}a \pm \frac{1}{2}\sqrt{a^2 - 4b}$.

If $a^2 - 4b < 0$, then the situation is simple. In that case, there are two complex conjugate roots. One can compute the real part $-\frac{1}{2}a$ and the imaginary part $\frac{1}{2}i\sqrt{4b - a^2}$ separately, without having to deal with numerical cancellation errors. From these two values, both complex conjugate roots can be constructed.

If $a^2 - 4b \geq 0$, then there can be numerical cancellation. In that case there are two real solutions. One of these real solutions is computed by selecting the most accurate solution as follows: if $a < 0$, then the solution $z_1 = -\frac{1}{2}a + \frac{1}{2}\sqrt{a^2 - 4b}$ is selected, else the solution $z_1 = -\frac{1}{2}a - \frac{1}{2}\sqrt{a^2 - 4b}$ is selected. The other real solution is computed as $z_2 = b/z_1$.

4 Software implementation and test suite

For testing purposes, an extensive test suite is developed, which can be used to generate polynomials, based on known (possibly randomly generated) roots, and generated at quadruple precision to be sure that the computation of coefficients from a given set of roots does not introduce irreversible significant loss of accuracy. Also a hand-selected set of polynomials is used to emphasize on specific "difficult" configurations of roots. Below follows a list of all software components, used in testing the algorithm:

MR An implementation of the well-established algorithm by K. Madsen and J.K. Reid, see [8]. This algorithm was ported from Fortran to Java. The Java version was used already for several years in several different applications without any issues and its output was thoroughly compared with the original Fortran code. This is used as a reference, because it is one of the most accurate general purpose polynomial root finders currently available for standard limited precision CPU architectures, which also has a good performance in terms of computational speed.

STR The original implementation, provided by Strobach, in Fortran90. This software is used to show that certain quartics are not handled correctly by the original algorithm of Strobach. Inspection of the code of this algorithm shows that it exactly implements the described algorithm of [5], so the issues really are in the algorithm and not in this particular Fortran-implementation.

WOE The implementation of the improved algorithm. This is written in Java and in C. The performance tests are done in Java in order to have a honest comparison with the **MR** algorithm. Functionally, there is no difference between the C code and the Java code, both produce exactly the same results, up to the last bit of accuracy.

QRT A suite of hand-selected quartics to test some "difficult" configurations of roots. All polynomials, tested by Orellana and De Michele [12] are included in this set. Besides that, some additional polynomials are added to this set. The individual polynomials are given further below, in the discussion of the results.

TST A test suite, which can be used to generate polynomials and which has the following features:

- Generation of polynomials from a given factor and a given set of roots, which can be real or conjugate complex pairs.
- Generation of polynomials from a given set of coefficients.
- Random generation of large numbers of polynomials, either based on generated roots, or based on generated coefficients. Distributions of roots or coefficients can be selected and roots can also be discretized to increase the chance of generation of multiple roots.
- Automated checking of accuracy, based on given roots and computed roots.

- Generation of polynomials is done with 106-bit quadruple precision. All given roots, factors, or coefficients are provided with 106-bit precision, and all computations in order to obtain the final set of coefficients also are done with 106-bit precision. These quadruple precision coefficients are converted to standard 53-bit double precision, before they are passed to the solver to be tested.

Two types of tests are performed. One type of tests is solving hand-selected quartics and comparing the results with the known values and with the results of other solvers. The other type of tests is to generate large numbers of randomly generated quartics with certain properties and comparing the computed roots with the generated roots, leading to an observed error for each computed root. The errors in turn are compared to a theoretical attainable error bound. The next section defines how the error bound is computed.

4.1 Maximum attainable numerical accuracy of a single root

How well can the roots of a polynomial be determined in a computer with limited precision, even when the applied root-finding method is perfect? The floating point precision leads to an absolute bound on the accuracy of the roots.

The analysis, given here, is for general polynomials over $\mathbb{C}$. Assume that z_0 is a root of multiplicity 1 of the n^{th} degree polynomial

$$P(z) = \sum_{k=0}^{n} p_k \, z^k \, . \tag{35}$$

Now assume that the j^{th} coefficient p_j is varied. It is straightforward to compute the sensitivity of the root z_0, relative to variation in p_j.

Using $P(z_0) = 0$, one can write

$$p_j \;=\; -\sum_{k=0,\, k\neq j}^{n} p_k \, z^{k-j} \, . \tag{36}$$

Using the fact that $(k-j)$ equals zero when $k = j$, taking the derivative, with respect to z, leads to

$$\frac{\partial p_j}{\partial z} \;=\; -\sum_{k=0,\, k\neq j}^{n} (k-j)\, p_k \, z^{k-j-1} \;=\; -\sum_{k=0}^{n} (k-j)\, p_k \, z^{k-j-1} \, . \tag{37}$$

By factoring out the factor z^{-j} one gets a nice and simple result:

$$\frac{\partial p_j}{\partial z} \;=\; -z^{-j} \sum_{k=0}^{n} (k-j)\, p_k \, z^{k-1} \;=\; -z^{-j}(P'(z) - jP(z)) \, . \tag{38}$$

At the single root $z = z_0$, the polynomial $P(z)$ equals 0, but the derivative $P'(z)$ is non-zero:

$$\left.\frac{\partial p_j}{\partial z}\right|_{z=z_0} \;=\; -z_0^{-j} P'(z_0) \, . \tag{39}$$

Taking the reciprocal gives the sensitivity of the root z_0, relative to changes in the j^{th} coefficient:

$$\frac{\partial z_0}{\partial p_j} \;=\; -\frac{{z_0}^j}{P'(z_0)}\,. \tag{40}$$

If the CPU precision is limited, due to the presence of a relative error, not greater than ϵ_{cpu}, then the absolute error in the coefficient p_j is bounded by $\epsilon_{\text{cpu}}\,|p_j|$. From equation (40), it follows that the contribution $E_{z0,j}$ to the total upper bound E_{z0} for the absolute error in the root z_0, due to uncertainty in the coefficient p_j, can be written as:

$$E_{z0,j} = \epsilon_{\text{cpu}}\,\frac{|p_j {z_0}^j|}{|P'(z_0)|} \tag{41}$$

If all contributions for all coefficients p_0 up to and including p_n to the total upper bound E_{z0} are combined into a single upper bound, then a simple expression is obtained, which easily can be evaluated:

$$E_{z0} \;=\; \sum_{k=0}^{n} E_{z0,k} \;=\; \sum_{k=0}^{n}\left(\epsilon_{\text{cpu}}\,\frac{|p_k {z_0}^k|}{|P'(z_0)|}\right) \;=\; \frac{\epsilon_{\text{cpu}}}{|P'(z_0)|}\;\sum_{k=0}^{n}|p_k {z_0}^k| \tag{42}$$

For a root $z_0 \neq 0$, the relative error ϵ_{z0} is bounded by

$$\epsilon_{z0} = \frac{\epsilon_{\text{cpu}}}{|z_0\,P'(z_0)|}\;\sum_{k=0}^{n}|p_k {z_0}^k| \tag{43}$$

Using equation (43), it is simple to determine error bounds for all simple roots of the polynomial equation. Any polynomial root solver, which produces relative errors in the roots, less than or equal to the bound given by equation (43), has done a perfect job.

4.2 Maximum attainable numerical accuracy of a multiple root

In a CPU architecture with limited floating point precision, there is a limit on the relative separation of two roots, which can with certainty be distinguished from a double root. Imagine that a polynomial equation has two very close roots z_1 and z_2 with upper absolute error bounds E_{z1} and E_{z2}, according to equation (42). If $|z_1 - z_2| < E_{z1} + E_{z2}$, then the polynomial might as well have a double root at $\frac{1}{2}z_1 + \frac{1}{2}z_2$. *It is not possible to distinguish between the double root and the two very close single roots!* In general a cluster of m very close roots cannot be distinguished from a root of multiplicity m, equal to the mean value of all separate roots. Detecting a cluster of roots is easy. Around each of the m roots z_c, $1 \leq c \leq m$, there is a disk of uncertainty $|z - z_c| \leq E_{zc}$. If the union of these disks forms a single connected set in the complex plane, then the associated roots form a cluster of roots, which best can be treated as a multiple root.

The error bound, given above, does not hold for multiple roots. If the root z_0 is a root of multiplicity $m > 1$, then all derivatives up to and including the $(m-1)^{\text{th}}$ derivate are

equal to 0. The analysis of the previous section, however, can be applied perfectly fine to the $(m-1)^{\text{th}}$ derivative of $P(z)$. For the latter polynomial, the root z_0 has multiplicity equal to 1. For very tight clusters of roots (which can easily be detected, see above), the $(m-1)^{\text{th}}$ derivative of $P(z)$ has a single root, equal to the average of all roots in the cluster. So, if the analysis of the previous section is performed on the $(m-1)^{\text{th}}$ derivative of $P(z)$, then the error bound, according to equation (42), is on the mean value of the m very close roots and not on the individual roots, and it is computed with $P(z)$ subsituted by the $(m-1)^{\text{th}}$ derivative of $P(z)$.

Bounds for the roots themselves also can be computed. A similar, but somewhat more involved derivation like the one given above, can be made for roots of general multiplicity m, see [13], section 7.4:

$$E_{z0} = \sqrt[m]{\frac{\epsilon_{\text{cpu}} \sum_{k=0}^{n} |p_k {z_0}^k|}{|P^{(m)}(z_0)|/m!}} \tag{44}$$

All m (possibly separate, but close) roots are in the disk of uncertainty $|z - z_0| \leq E_{z0}$, where $P^{(m)}(z)$ is the m^{th} derivative of $P(z)$. For $m = 1$, this again gives the result of equation (42).

Summarizing, for the estimates of multiple roots, which usually will be separate, but very close numbers, it is checked that all estimates are in a disk around the true multiple root, according to equation (44).

4.3 The error factor of a polynomial for a software solver

For each of the roots z_1, z_2, . . . z_n of an n^{th} degree polynomial P(z) one can define a ratio F_1, F_2, . . . F_N as follows:

$$F_i = \frac{|z_i - \hat{z}_i|}{E_{zi}} \qquad (1 \leq i \leq n), \tag{45}$$

where E_{zi} is the error bound for root z_i, according to (42) or (44), and $\hat{z}_i$ is the approximation of the root z_i, computed by the software for solving equations. The *error factor* for the polynomial P for a certain software solver can be defined as

$$F = max(F_1, F_2, ...F_n) \tag{46}$$

For an ideal piece of software, $F \leq 1$. Such a piece of software has errors in all computed roots, at most equal to the theoretical error bound. For testing the quartic equation solver, the error factor F is used as a measure for the performance of the algorithm.

4.4 Testing the quartic root solver

Testing of the quartic root solver is done in different ways.

- Generate a set of 10^9 random quartics. For each of the quartics, randomly select 4 roots z_1, z_2, z_3, and z_4, according to some random distribution, described below in

the section on the test results. Also generate a random factor f and compute the five coefficients of the quartic $f \cdot (z - z_1)(z - z_2)(z - z_3)(z - z_4)$. Do all these computations in 106 bit precision.

- For each of the generated quartics, convert the coefficients to 53-bit standard precision by rounding. Use the quartic solver, as described in this paper on the rounded quartics. Also use the MR-solver on the rounded quartics.

- Compute the error factor F, according to equation (46), from the difference between the generated (given) roots, and the roots computed by the solver from the given 53-bit coefficients of the quartic. Also compute the error factor for the MR-solver.

- Also keep track of the time, needed by the solver to solve all quartics. The test code is written in such a way, that the time needed for generation of the quartics, and all post-processing and comparisons of the solutions, is not taken into account in the timing. Hence, all quartics first are generated and stored in memory, then they are solved in a tight loop and the computed roots are stored for each quartic, and finally, for each quartic, the computed roots are compared with the generated roots. The same setup also is used for the MR-solver.

- Besides using the quartics with discretized roots, similar tests are done with roots, which are not discretized. This leads to somewhat more random roots, with much fewer double roots, but more very close roots. The same random distributions are used again.

- Besides using the above-mentioned randomly generated quartics, also a suite of hand-selected quartics is used for testing the solver. The quartics in this suite are selected on the basis of their perceived difficult properties, such as tight clusters of roots, multiple roots, high dynamic range of roots.

- The original Fortran code from Strobach is used to show that some quartics are not solved correctly and it is shown where his algorithm fails and how these situations are taken care of in the algorithm, presented in this paper.

5 Result of tests

Here, results of tests of the solver are presented. First, several sets of random quartics are described in more detail and it is explained what accuracy can be obtained for solving these quartics, and next, the results of a suite of hand-selected quartics is presented.

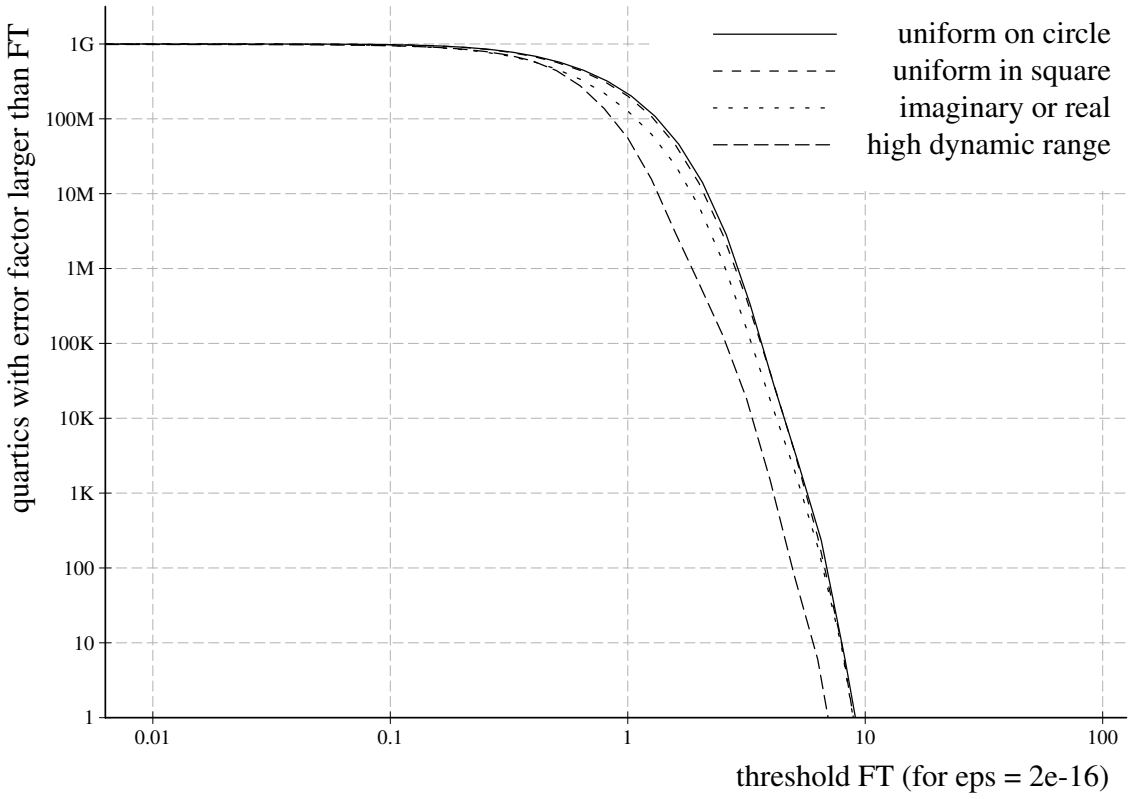
5.1 Result with large numbers of randomly generated quartics

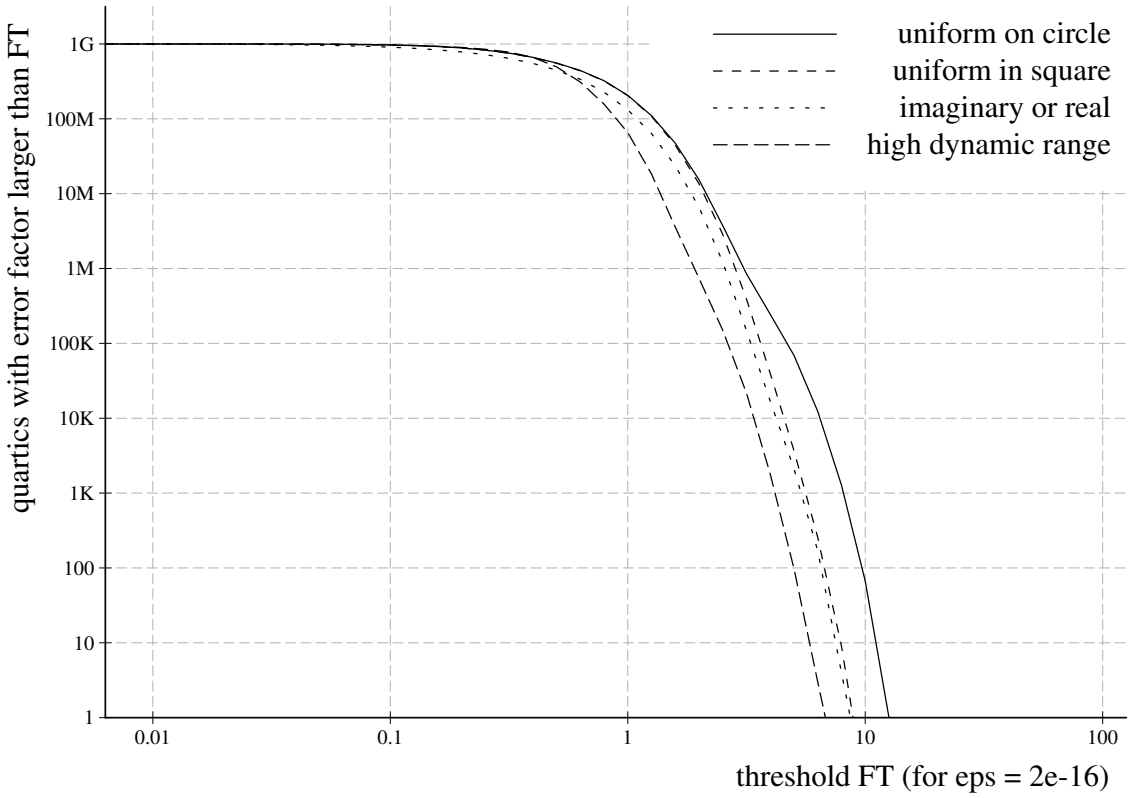
In this test, random quartics are generated, with different distributions of roots. For each distribution, one set of 10^9 quartics is generated. In all sets, the generated equations either have 0 complex roots, 2 complex roots, or 4 complex roots, each of these configurations having a probability of $\frac{1}{3}$. Complex roots come in conjugate pairs. In each set, roots are generated, according to one of the random distributions, described below. After generating the roots, both the real and imaginary part are rounded to the nearest multiple of 0.001, before the coefficients of the polynomial are computed. This increases the chance of generating double roots. For the used distributions, the chance of an exact double root is appr. 1 out of 10000 for each pair of real roots. Summarizing, the following distributions are used:

- Random roots, uniformly distributed on the circle $|z| = 5$, or on the real axis with $-5 \leq \Re(z) \leq 5$.
- Uniformly distributed random roots inside or on a square with $-5 \leq \Re(z) \leq 5$, and $-5 \leq \Im(z) \leq 5$, or real roots with $-5 \leq \Re(z) \leq 5$.
- Random roots, uniformly distributed on the imaginary axis with $-5 \leq \Im(z) \leq 5$, or on the real axis with $-5 \leq \Re(z) \leq 5$.
- Roots, which are generated, according to the second distribution (inside or on a square), but after generation and rounding, they are multiplied by a factor 10^n, with n an integer value uniformly randomly selected from the set {-20, -19, ..., 19, 20}. This assures that quartics are generated with roots in a wide dynamic range.

For each of the generated quartics the error factor F is computed for a machine precision $\epsilon_{\text{cpu}} = 2 \cdot 10^{-16}$, according to equation (46). The plot below shows a graph for all four distributions, described above, with the number of quartics having an error factor F, larger than a threshold value factor F_{T}. This plot shows that appr. 80% to 90% of all generated quartics have an error factor less than 1, depending on the distribution of the roots. Even more important, though, is that the error factor is less than 10 for all quartics, and hence never more than 1 digit of precision is lost, relative to the theoretically attainable accuracy.

A similar test was repeated, but now without rounding to the nearest multiple of 0.001. This test shows similar results, with only a very low fraction of quartics with loss of just over 1 digit of accuracy, relative to the theoretically attainable accuracy.


quartics with error factor larger than FT
1G
100M
10M
1M
100K
10K
1K
100
10
1
0.01
0.1
1
10
100
threshold FT (for eps = 2e-16)
uniform on circle
uniform in square
imaginary or real
high dynamic range


quartics with error factor larger than FT
1G
100M
10M
1M
100K
10K
1K
100
10
1
0.01
0.1
1
10
100
threshold FT (for eps = 2e-16)
uniform on circle
uniform in square
imaginary or real
high dynamic range

The solver has no issues with multiple roots. Roots with multiplicity m have a number of correct digits, which is roughly $\frac{1}{m}^{\text{th}}$ part of the number of correct digits of single roots. Sometimes the number of correct digits is much larger for double or triple roots, but this only happens when the roots have nice "round" values, like 1, or 1000. Such numbers can be represented exactly and also products of four of such numbers still can be represented exactly and that leads to exact results. This also is observed in the manually selected quartics, given in the next section. When the roots are replaced by values like 1.0001234, then this effect is lost.

5.2 Result with suite of hand-selected quartics

The quartics, used in this test are tabulated, together with why the quartic was selected and what result is obtained. The table can be found on the next page. The quartics are based on those, used by Orellana et al., see [12], at the top of section 3. For some quartics, with multiple roots, Orellana et al. report unrealistically good results. In order to assure that these good results are not due to a lucky effect of exact representation of numbers with few digits, slightly modified values were used in the test suite, given below. The quartics, which were made less "round" are the ones in cases 14, 15, 20, and 21.

The result of the tests with these hand-selected quartics is very well. All of these quartics have errors, well within the theoretical bounds of equations (42) and (44). All single roots have relative errors less than 10^{-15}, except the ones of cases 2, 11, and 17. These roots are very close and hence are ill conditioned. The roots of case 2 all have a relative error less than 10^{-6}. The roots of case 11 have a relative error of appr. $3.5 \cdot 10^{-13}$, and the roots of case 17 have errors of appr. $2.5 \cdot 10^{-7}$. All these errors are well within the theoretical bound of equation (42).

Double roots have accuracies of 7 to 8 digits, and even the quadruple root of case 14 has almost 8 correct digits. According to the theoretical bounds this should be much lower (only appr. 4 digits). Most likely, there still is the effect of lucky rounding. The triple root of case 15 has a little over 5 digits of accuracy, and this is close to the theoretical bound of equation (44).

5.3 Comparison with Strobach solver and Orellana solver

The solver, presented in this paper, does not use polishing of solutions or polishing of quadratic factors after the analytic computation, based on the LDL^{T} decomposition. Both Strobach [5] and Orellana et al. [12] use polishing after the analytic steps. Polishing, as presented by Orellana et al. can also be added to the algorithm, presented here, but it is not needed. The solver loses at most just over 1 digit of accuracy in some unlucky cases. The possible slight improvement is not worth the computational cost and the added complexity of the code and there even is the risk that the polishing adds new unforeseen numerical problems. Strobach's solver fails for several quartics, and Orellana's solver has inaccurate results for several quartics (sometimes loss of more than 10 digits), which needs repairing by means of polishing, using Newton iterations.

case	**roots**	**reason**
1	10^9, 10^6, 10^3, 1	large dynamic range
2	2.003, 2.002, 2.001, 2	very close real roots
3	10^{53}, 10^{50}, 10^{49}, 10^{47}	Large real values
4	10^{14}, 2, 1, -1	One large, three small roots
5	$2{\cdot}10^7$, 10^7, 1, -1	Two large real, two small real
6	10^7, -10^6, $1 \pm i$	Two large real, two small complex
7	-7, -4, $-10^6 \pm 10^5 i$	Two small real, two large complex
8	10^8, 11, $10^3 \pm i$	Large dynamic range, real and complex
9	$10^7 \pm 10^6 i$, $1 \pm 2i$	Two large, two small complex
10	$10^4 \pm 3i$, $-7 \pm 10^3 i$	Four complex, mixed magnitude
11	$1.001 \pm 4.998i$, $1 \pm 5.001i$	Two clusters of close complex roots
12	$10^3 \pm 3i$, $10^3 \pm i$	Cluster of four fairly close roots
13	$2 \pm 10^4 i$, $1 \pm 10^3 i$	Four complex, small real, large imag.
14	1000.1234, 1000.1234, 1000.1234, 1000.1234	Four equal roots
15	1000.1234, 1000.1234, 1000.1234, 10^{-15}	Three equal roots plus small root
16	$1 \pm 0.1i$, $10^{16} \pm 10^7 i$	Four complex, large dynamic range
17	10000, 10001, 10010, 10100	Four nearly equal real roots
18	$40000 \pm 300i$, $30000 \pm 7000i$	Four complex, moderately large
19	10^{44}, 10^{30}, 10^{30}, 1	Huge dynamic range and double root
20	$1.0001234{\cdot}10^{14}$, $1.0001234{\cdot}10^7$, $1.0001234{\cdot}10^7$, 1.0001234	Large dynamic range and double root
21	$1.0001234{\cdot}10^{15}$, $1.0001234{\cdot}10^7$, $1.0001234{\cdot}10^7$, 1.0001234	Large dynamic range and double root
22	10^{154}, 10^{152}, 10, 1	Dynamic range , max. representable
case	**quartic coefficients**	**reason**
23	$a = 1, b = 1, c = \frac{3}{8}, d = 10^{-3}$	d_2 close to zero
24	$a = -\left(1 + \frac{1}{S}\right),\ b = \frac{1}{S} - S^2,$ $c = S^2 + S,\ d = -S$ with $S = 10^{30}$	Widely spaced coefficients

5.4 Performance in terms of computational speed

The performance of the solver in terms of computational speed, presented in this paper, is comparable to the performance of Strobach's solver or the solver of Orellana et al. This also was expected, because all three algorithms are based on the same underlying principle and are quite similar. Compared to an efficient implementation of the general polynomial solver, based on the Madsen and Reid algorithm, see [8], the three solvers are much faster, a speedup factor of 5 is achievable without problems.

With careful optimization, and tuning compiler or JVM-settings, probably a little more speedup can be achieved, but in this work, this was not further pursued. A standard JVM (Oracle Hotspot, Java Runtime Environment 11.0.10) was used, out of the box, without any tweaking of parameters.

5.5 Possible improvements and optimizations

Tests were performed with Orellana's algorithm (see [12], section 2.2) for computing the solutions of the depressed cubic equation (4). This algorithm looks attractive, because of its great simplicity, but this gave less satisfactory results in terms of accuracy. Strobach's algorithm, using so-called 'bi-iterations', see [5], section 3.1, performs exceptionally well in terms of accuracy, while still being reasonably fast. Tests with billions of randomly generated depressed cubics with many different distributions of g and h show that the relative error never exceeds 10^{-15} on a machine with $\epsilon_{\text{CPU}} = 2 \cdot 10^{-16}$. Using the algorithm, presented in [12], but without the Newton steps at the end, only leads to a modest speedup of 10% to 20% of the total time for solving a quartic, but at the cost of severely increased errors and the need to polish solutions afterwards. So, it was decided to stick to Strobach's algorithm for solving the depressed cubic equation (4).

Most modern CPU's have a so-called fused multiply and add instruction (FMA), which computes $a * b + c$ for any given floating point numbers a, b, and c, using exact arithmetic for the complete computation and only rounding at the end of the computation. Standard arithmetic has rounding after computing $a * b$ and again after adding c. Using FMA-instructions, there may be some additional accuracy, and it also may make the code slightly faster, but no spectacular improvements should be expected. Using of FMA-instructions does not introduce the risk of unforeseen numerical issues and may be a more fruitful direction of research than using polishing of roots at the end of the algorithm.

5.6 Conclusions

A nice basis had been laid by Strobach with the introduction of the LDL^{T} quartic solver. With the addition of runtime accuracy analysis, a really accurate solver is obtained, which is nearly as fast as the currently available analytical solvers and at the same time has an accuracy very close to the theoretically attainable accuracy, also in all kinds of corner cases, where many other solvers have a hard time getting good results. Probably, there is no other quartic equation solver, which can beat the one, presented in this paper. This is demonstrated by the accompanying code and the extensive test suite, which addresses all known identified corner cases.

References

[1] H. E. Salzer, "A note on the solution of quartic equations," *Math. Comput.*, vol. 14, no. 71, pp. 279–281, 1960.

[2] N. Flocke, "Algorithm 954: An accurate and efficient cubic and quartic equation solver for physical applications," *ACM Trans. Math. Softw.*, vol. 41, pp. 30:1–30:24, 2015.

[3] S. L. Shmakov, "A universal method of solving quartic equations," *Int. J. Pure Appl. Math.*, vol. 71, no. 2, pp. 251–259, 2011.

[4] P. Strobach, "The fast quartic solver," *J. Comput. Appl. Math.*, vol. 234, no. 10, pp. 3007–3024, 2010.

[5] P. Strobach, "The low-rank ldlt quartic solver," *Internal Technical Report*, pp. 1–26, 2015. AST-Consulting Inc., Bahnsteig 6, 94133 Röhrnbach, Germany (www.ast-consulting.net), https://doi.org/10.13140/2.1.3955.7440.

[6] M. D. Yacoub and G. Fraidenraich, "A solution to the quartic equation," *Math. Gaz.*, vol. 96, no. 536, pp. 271–275, 2012.

[7] M. A. Jenkins and J. F. Traub, "A three-stage algorithm for real polynomials using quadratic iteration," *SIAM J. Numer. Anal.*, vol. 7, no. 4, pp. 545–566, 1970.

[8] K. Madsen and J. K. Reid, *Fortran subroutines for finding polynomial zeros, Report HL 75/1172(C13)*. Computer Science and Systems Divisions, A.E.R.E Harwell, Oxfordshire, feb 1975.

[9] D. A. Bini and G. Fiorentino, "Design, analysis, and implementation of a multiprecision polynomial rootfinder," *Numer. Algorithms*, vol. 23, pp. 127–173, jun 2000.

[10] R. Descartes, *On the construction of solid and supersolid problems*, vol. III. Dover, 1954 [1637]. ISBN 0-486-60068-8, JFM 51.0020.07.

[11] L. Euler, *Of a new method of resolving equations of the fourth degree*. Springer-Verlag, 1984 [1765]. Elements of Algebra, ISBN 978-1-4613-8511-0, Zbl 0557.01014.

[12] A. G. Orellana and C. de Michele, "Algorithm 1010: Boosting efficiency in solving quartic equations with no compromise in accuracy," *ACM Trans. Math. Softw.*, vol. 46, no. 2, 2020. Article 20 (May 2020), https://doi.org/10.1145/3386241.

[13] J. H. Wilkinson, *The algebraic eigenvalue problem*. Clarendon Press, 1965. Oxford.