Annales Mathematicae et Informaticae
***33** (2006) pp. 3–13*
http://www.ektf.hu/tanszek/matematika/ami

Multiple root finder algorithm for Legendre and Chebyshev polynomials via Newton's method

Victor Barrera-Figueroa[a], Jorge Sosa-Pedroza[b], José López-Bonilla[c]

[abc]Instituto Politécnico Nacional, Escuela Superior de Ingeniería Mecánica y Eléctrica, Sección de Estudios de Postgrado e Investigación
[a]e-mail: vbarreraf@ipn.mx; [b]e-mail:jsosa@ipn.mx; [c]e-mail:jlopezb@ipn.mx

Submitted 23 June 2006; Accepted 31 October 2006

Abstract

We exhibit a numerical technique based on Newton's method for finding all the roots of Legendre and Chebyshev polynomials, which execute less iterations than the standard Newton's method and whose results can be compared with those for Chebyshev polynomials roots, for which exists a well known analytical formula. Our algorithm guarantees at least nine decimal correct ciphers in the worst case, however, when comparing with Chebyshev roots given by its formula, even eighteen decimal correct ciphers are achieved in several roots, in the best case. As a comparison guide the results are collated with those gotten by MATLAB.

Keywords: Newton's method, Legendre polynomials, Chebyshev polynomials, multiple root finder algorithm.

1. Introduction

Legendre polynomials (see [1, 2, 3, 4]) as well as Chebyshev (see [1, 2, 3, 4, 5]) ones has found countless applications in all branches of engineering and science, among which the most representatives include calculation of quadratures, electromagnetics and antenna applications, solutions for potential theory and for Schrödinger's equation, aerodynamics and mechanics applications, etc. This is due mainly because the use of their roots allows us to solve a specific problem with the best efficiency.

These polynomials are very similar to each other, not only in the form of their respective differential equation, but also in the numerical values of their roots.

However, sometimes it is better the use of Legendre polynomials instead of Chebyshev ones because by using their roots, one can reach the most optimized solution. For instance, when calculating a quadrature, Gauss established that the best way for obtaining the minimum error in a numerical integration is by dividing the integration domain in agreement with the way Legendre roots are distributed on their own domain.

So, because of the wide use of Legendre and Chebyshev roots, this paper presents a multiple root finder algorithm, which is expected to be a useful tool, not only in engineering work but also in numerical and mathematical analysis. This algorithm is based in the classical Newton's method (see [2]), however, we have made some modifications in order to find all the roots of an specific polynomial by using the improved Newton's method (see [2]).

2. Newton's method

Newton's process is a numerical tool used to find the zero x_e of a real-valued function $f(x)$, continuously differentiable and whose derivative does not vanish at $x = x_e$. The method consists in proposing an initial root x_0 which must be in the neighborhood of x_e, as shown in Figure 1. Once made this, we draw a tangent line to $f(x)$ at $x = x_0$, and determine its intersection with the x axis. The equation for the line is:

$$y = f(x_0) + f'(x_0)(x - x_0), \tag{2.1}$$

which means that locally, in the small neighborhood of x_0, $f(x)$ can be considered as a linear equation, even if $f(x)$ is not linear. We should notice that this equation corresponds to the truncate Taylor series with center at $x = x_0$ by neglecting the remainder term. The intersection, named x_1, is then:

$$x_1 - x_0 = -\frac{f(x_0)}{f'(x_0)}, \tag{2.2}$$

Now, x_1 is closer to the real zero x_e, and it will be used to draw the next tangent line to $f(x)$ at $x = x_1$ which produces the intersection x_2. So, in the n_{th} iteration, the intersection of the tangent line with the x axis will be [2]:

$$x_n = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})}, \quad n = 1, 2, 3, \ldots \tag{2.3}$$

After a number of iterations x_n will be close enough to x_e. In this way, once we have established the allowed error ϵ for the zero, we can set the condition for stopping the method, which is reached when:

$$\mid x_n - x_{n-1} \mid \leqslant \epsilon. \tag{2.4}$$

Newton's technique will usually converge provided the initial guess is close enough to true root. Furthermore, for a zero of multiplicity 1, the convergence

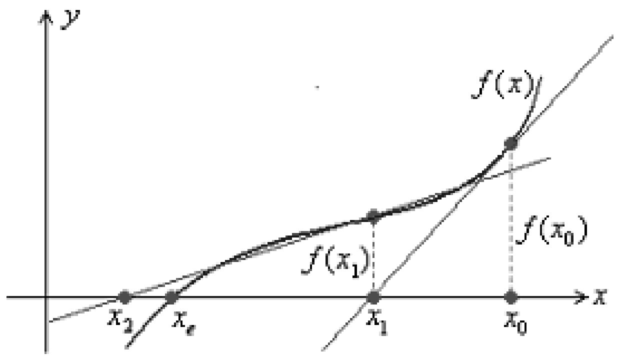

Figure 1: Tangent lines to $f(x)$ at $x = x_1$.

is at least quadratic in a neighborhood of x_e, which intuitively means that the number of correct digits roughly at least doubles in every step.

Newton's method acquire a bigger convergence if in Taylor series we consider the quadratic term:

$$y = f(x_0) + f'(x_0)(x - x_0) + \frac{1}{2} f''(x_0)(x - x_0)^2, \tag{2.5}$$

that is, in the small neighborhood of x_0, $f(x)$ can be considered as a quadratic equation, even if $f(x)$ is not quadratic. The intersection x_1 between (2.5) and the x axis can be calculated from:

$$x_1 - x_0 = -\frac{f(x_0)}{f'(x_0) + \frac{1}{2} f''(x_0)(x_1 - x_0)}, \tag{2.6}$$

by substituting (2.2) into (2.6) we get:

$$x_1 - x_0 = -\frac{2f(x_0)f'(x_0)}{2\left[f'(x_0)\right]^2 - f(x_0)f''(x_0)}. \tag{2.7}$$

So, in the n_{th} iteration, we have the next recursive formula (see [2]):

$$x_n = x_{n-1} - \frac{2f(x_{n-1})f'(x_{n-1})}{2\left[f'(x_{n-1})\right]^2 - f(x_{n-1})f''(x_{n-1})}, \quad n = 1, 2, 3, \dots \tag{2.8}$$

This is the improved Newton's method which converges faster than the original one. As well as we have to know the first derivative of $f(x)$ in both methods, in the improved one, we must know the second derivative of $f(x)$ in each point. This could be a problem if $f(x)$ is not an analytical formula but a set of data points, in which both derivatives could be calculated from standard numerical techniques.

3. Multiple root finder algorithm based on Newton's process

The application of Newton's method to a polynomial in the quest for all its roots could be easily implemented if we express it according to the fundamental theorem of Algebra:

$$f(x) = k(x - x_1)(x - x_2)\dots(x - x_N), \tag{3.1}$$

where N is the polynomial order, k is a proportionality constant and $\{x_1, x_2, \dots, x_N\}$ are all the polynomial roots, which are not necessarily put in order. In the following, the subindex j in the root x_j represents the number of root and it should not be confused with the number of iterations used in the previous sections which will be omitted from here. By applying Newton's algorithm to the polynomial (3.1) for searching the first root x_1, we have the next recursive relation:

$$x_1 = x_1 - \frac{f(x)}{f'(x)}. \tag{3.2}$$

Once x_1 is found, we built the polynomial $g(x)$ of order $N - 1$ which has not x_1 as one of its roots:

$$g(x) = \frac{f(x)}{x - x_1} = k(x - x_2)(x - x_3)\dots(x - x_N), \tag{3.3}$$

and by applying Newton's technique to $g(x)$ we determine the next root x_2 and so on. So, in the k_{th} step, we built the following polynomial:

$$g(x) = \frac{f(x)}{\prod_{i=1}^{k-1}(x - x_i)}, \tag{3.4}$$

and, according to Newton's method, the k_{th} root is:

$$x_k = x_k - \frac{g(x_k)}{g'(x_k)}, \tag{3.5}$$

which implies the calculation for the derivative of $g(x)$ which could not be so evident because of the need to find the derivative of the product term. Such derivative is:

$$\begin{aligned}\frac{\mathrm{d}}{\mathrm{d}x}\prod_{i=1}^{k-1}(x - x_i) &= \left[\frac{1}{x - x_1} + \frac{1}{x - x_2} + \cdots + \frac{1}{x - x_{k-1}}\right]\prod_{i=1}^{k-1}(x - x_i) = \\ &= \prod_{i=1}^{k-1}(x - x_i)\sum_{i=1}^{k-1}\frac{1}{x - x_i},\end{aligned} \tag{3.6}$$

therefore the derivative for $g(x)$ is:

$$g'(x) = \frac{1}{\prod_{i=1}^{k-1}(x-x_i)}\left[f'(x) - f(x)\sum_{i=1}^{k-1}\frac{1}{x-x_i}\right]. \tag{3.7}$$

By substituting (3.7) and (3.4) into (3.5) we reach the recursive relation for getting all the roots for the polynomial $f(x)$:

$$x_k = x_k - \frac{f(x_k)}{f'(x_k) - f(x_k)\sum_{i=1}^{k-1}\frac{1}{x-x_i}}, \quad k = 1, 2, \ldots, N. \tag{3.8}$$

On the basis of (3.8) we express the multiple root finder algorithm with the following diagram:

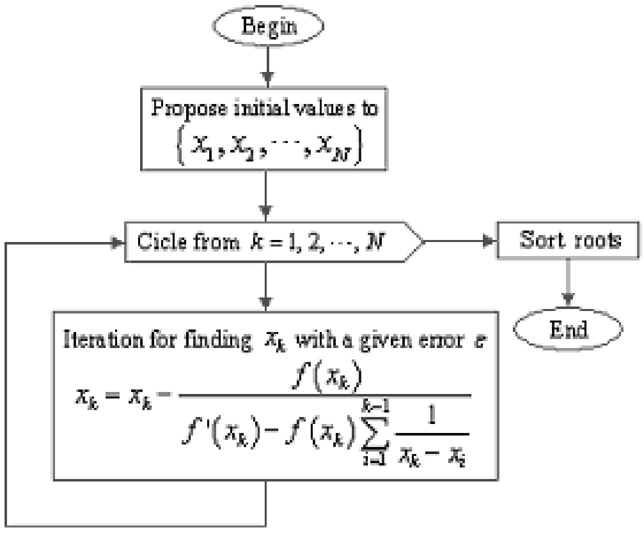
Begin
Propose initial values to $\{x_1, x_2, \cdots, x_N\}$
Cicle from $k = 1, 2, \cdots, N$
Sort roots
End
Iteration for finding x_k with a given error ε
$x_k = x_k - \frac{f(x_k)}{f'(x_k) - f(x_k)\sum_{i=1}^{k-1}\frac{1}{x_k - x_i}}$

Figure 2: Multiple root finder algorithm.

In order to sort all the roots given by the multiple root finder process, we can introduce the well known bubble sort method which by successively interchanging the N roots can provide us the list of expected results. The bubble sort method is drawn in the following diagram:

4. Multiple root finder algorithm based on improved Newton's method

In order to improve the algorithm's convergence, we can introduce Newton's technique in the quest for all the polynomial roots. In first term we look at the

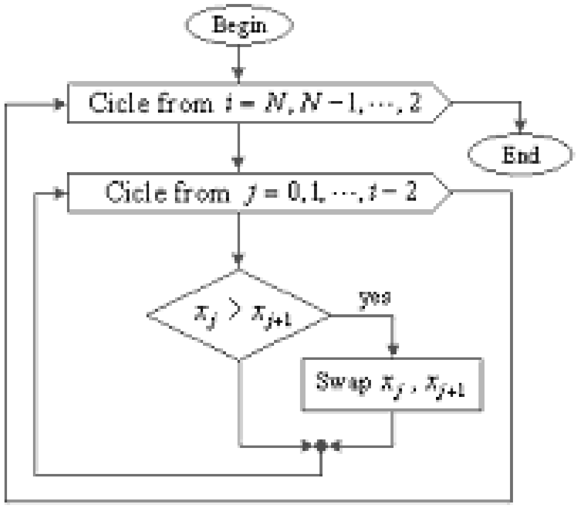
Begin
Cicle from $i = N, N-1, \cdots, 2$
End
Cicle from $j = 0, 1, \cdots, i-2$
$x_j > x_{j+1}$
yes
Swap x_j, x_{j+1}

Figure 3: Bubble sort method for putting in order the polynomial's roots.

root x_1 of the polynomial (3.1) with the recursive relation (2.8). Once x_1 is found, we built the polynomial $g(x)$, (3.3). By applying improved Newton's method to $g(x)$ we found the next root x_2 and so on. So, in the k_{th} step, we built the polynomial (3.4), from which the k_{th} root is:

$$x_k = x_k - \frac{2g(x_k)g'(x_k)}{2\left[g'(x_k)\right]^2 - g(x_k)g''(x_k)}. \tag{4.1}$$

Iterative formula (4.1) implies the calculation for the second derivative of $g(x)$ which could a hard task, however, the result is not as difficult as one could expect:

$$g''(x) = \frac{f''(x) - 2f'(x)\sum\limits_{i=1}^{k-1}\frac{1}{x-x_i} + f(x)\left[\sum\limits_{i=1}^{k-1}\frac{1}{(x-x_i)^2} + \left(\sum\limits_{i=1}^{k-1}\frac{1}{x-x_i}\right)^2\right]}{\prod\limits_{i=1}^{k-1}(x-x_i)}. \tag{4.2}$$

So by substituting (3.4),(3.7) and (4.2) into (4.1) we get:

$$x_k = x_k - \frac{2f(x_k)B(x_k)}{B^2(x_k) + \left[f'(x_k)\right]^2 - f(x_k)\left[f''(x_k) + f(x_k)\sum\limits_{i=1}^{k-1}\frac{1}{(x_k-x_i)^2}\right]}, \tag{4.3}$$

$$B(x_k) = f'(x_k) - f(x_k)\sum_{i=1}^{k-1}\frac{1}{x_k - x_i}, \quad k = 1, 2, \ldots, N.$$

Notice the similarity between (4.1) and (4.3) where $f(x_k)$ is similar to $g(x_k)$, and $g'(x_k)$ is analogous to $B(x_k)$. Obviously both formulas are not completely analogous, but the existent similarity is notorious. On the basis of (4.3) we express the improved multiple root finder algorithm with the following diagram, Figure 4, again, the bubble sort method could be used in order to sort the obtained roots:

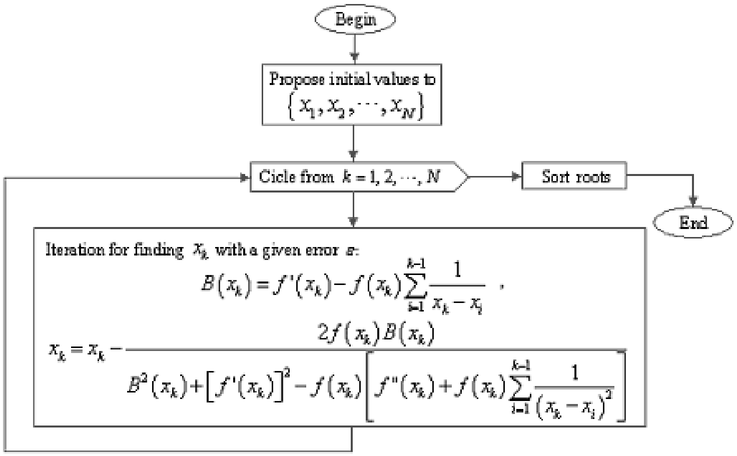
Begin
Propose initial values to $\{x_1, x_2, \cdots, x_N\}$
Cicle from $k = 1, 2, \cdots, N$
Sort roots
End
Iteration for finding x_k with a given error ε:
$B(x_k) = f'(x_k) - f(x_k)\sum_{i=1}^{k-1}\frac{1}{x_k - x_i}$,
$x_k = x_k - \frac{2f(x_k)B(x_k)}{B^2(x_k) + [f'(x_k)]^2 - f(x_k)\left[f''(x_k) + f(x_k)\sum_{i=1}^{k-1}\frac{1}{(x_k - x_i)^2}\right]}$

Figure 4: Improved multiple root finder algorithm.

5. Legendre and Chebyshev polynomials roots calculation

In this section we present the results reached by both algorithms applied to Legendre and Chebyshev polynomials; also, it is shown the number of performed iteration. Such results are compared with those gotten by MATLAB. In the case of Chebyshev's roots, we provide the analytical results gotten by the formula [2]:

$$x_k = \cos\left(\frac{2k-1}{2N}\pi\right), \quad k = 1, 2, \ldots, N, \tag{5.1}$$

where N is the polynomial degree. Legendre polynomials $P_n(x)$ are the solution of the Legendre differential equation:

$$(1 - x^2)\frac{\mathrm{d}^2 P_n(x)}{\mathrm{d}x^2} - 2x\frac{\mathrm{d}P_n(x)}{\mathrm{d}x} + n(n+1)P_n(x) = 0, \quad n = 0, 1, 2, \ldots\ , \tag{5.2}$$

while Chebyshev polynomials $T_n(x)$ are the solution for its respective differential equation:

$$(1-x^2)\frac{\mathrm{d}^2T_n(x)}{\mathrm{d}x^2} - x\frac{\mathrm{d}T_n(x)}{\mathrm{d}x} + n^2T_n(x) = 0, \quad n = 0,1,2,\dots\ . \tag{5.3}$$

Both $P_n(x)$ and $T_n(x)$ can be defined by mean of recursive relations, which is an important issue in the numerical point of view. For Legendre polynomials, the recursive relations are:

$$P_0(x) = 1,\ P_1(x) = x,\ P_{n+2} = \frac{2n+3}{n+2}xP_{n+1} - \frac{n+1}{n+2}P_n,\ n = 0,1,2,\dots\ , \tag{5.4}$$

while for Chebyshev polynomials the recursive relations are:

$$T_0(x) = 1,\ T_1(x) = x,\ T_{n+2} = 2xT_{n+1} - T_n, \quad n = 0,1,2,\dots\ , \tag{5.5}$$

Such polynomials are plotted in Figure 5. It is notable how the roots are clustered in the ends of the domain $[-1,1]$ in both polynomials.

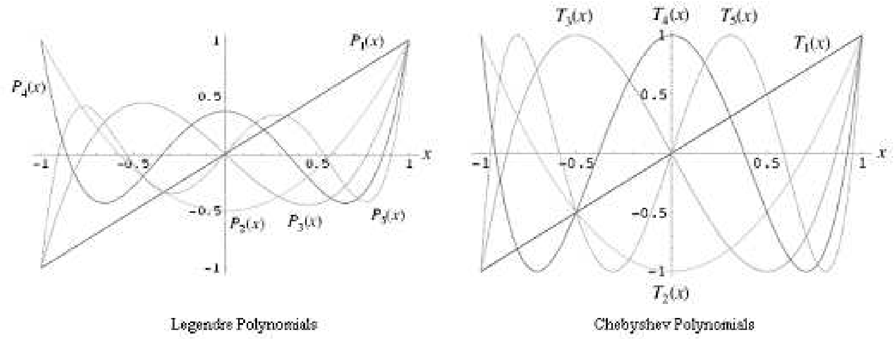

Figure 5: Legendre and Chebyshev polynomials.

In Table 1 are shown the results for $P_{19}(x)$ roots reached by both algorithms, with a permissible error of $\epsilon = 1 \times 10^{-12}$ for each of them.

In Table 2 the roots for $P_{19}(x)$ obtained using MATLAB are presented for comparison. In Table 3 are shown the results for $T_{19}(x)$ roots reached by both algorithms, with a permissible error of $\epsilon = 1 \times 10^{-12}$ for each of them.

In Table 4 the roots for $T_{19}(x)$ obtained using (5.1) and MATLAB are presented for comparison.

6. Conclusions

Both algorithms have shown to be effective in finding all the roots for a specific polynomial. Such polynomial must have all its roots real and with simple multiplicity. These conditions are satisfied by Legendre and Chebyshev polynomials.

k	Multiple root finder algorithm, x_k	Number of iterations	Improved multiple root finder algorithm, x_k	Number of iterations
1	-0.992406843843584352	2	-0.992406843843584350	3
2	-0.960208152134830018	12	-0.960208152134830020	4
3	-0.903155903614817901	15	-0.903155903614817900	7
4	-0.822714656537142819	6	-0.822714656537142820	4
5	0.720966177335229386	6	-0.720966177335229390	4
6	-0.600545304661680990	6	-0.600545304661680990	5
7	-0.464570741375960938	7	-0.464570741375960940	5
8	-0.316564099963629830	8	-0.316564099963629830	6
9	-0.160358645640225367	9	-0.160358645640225390	6
10	0.000000000000000000	10	0.000000000000000000	6
11	0.160358645640225367	10	0.160358645640225390	7
12	0.316564099963629830	11	0.316564099963629830	7
13	0.464570741375960938	11	0.464570741375960940	7
14	0.600545304661680990	12	0.600545304661680990	8
15	0.720966177335229386	11	0.720966177335229390	8
16	0.822714656537142819	11	0.822714656537142820	8
17	0.903155903614817901	10	0.903155903614817900	7
18	0.960208152134830018	8	0.960208152134830020	7
19	0.992406843843584352	8	0.992406843843584460	5

Table 1: Roots for $P_{19}(x)$ with both algorithms.

However, for other type of polynomials, another process is being developed in order to get not only their real roots but also their complex ones, taking into account their own multiplicity.

The first algorithm is faster than the second one, because it performs fewer operations than the improved one, which must calculate the second derivative and several sums, among other operations. However, the second algorithm executes less iteration than the first one, and this fact could be useful in finding the roots for polynomials of big order. Both algorithms provide us several correct ciphers when comparing with the results gotten by MATLAB, however, both are faster than the routines used by MATLAB in doing the same action.

It is expected that these algorithms can be employed as a part for several programs which must involve quadratures or interpolations (among other numerical issues) while looking for engineering or science solutions, because of its speed and ease for programming. We the authors recommend an error of $\epsilon = 1 \times 10^{-12}$ for each root, which has shown to provide correct results for all the possible polynomial degrees. However, for an error of $\epsilon = 1 \times 10^{-18}$ in some degrees, both algorithms perform a great number of iterations in which the loop becomes endless.

k	MATLAB, x_k
1	-0.992406843843584350
2	-0.960208152134830020
3	-0.903155903614817900
4	-0.822714656537142820
5	-0.720966177335229390
6	-0.600545304661680990
7	-0.464570741375960940
8	-0.316564099963629830
9	-0.160358645640225370
10	0.000000000000000000
11	0.160358645640225370
12	0.316564099963629830
13	0.464570741375960940
14	0.600545304661680990
15	0.720966177335229390
16	0.822714656537142820
17	0.903155903614817900
18	0.960208152134830020
19	0.992406843843584350

Table 2: Roots for $P_{19}(x)$ with MATLAB.

k	Multiple root finder algorithm, x_k	Number of iterations	Improved multiple root finder algorithm, x_k	Number of iterations
1	-0.996584493006669847	2	-0.996584493006669850	3
2	-0.969400265939330374	12	-0.969400265939330370	5
3	-0.915773326655057396	15	-0.915773326655057400	7
4	-0.837166478262528546	4	-0.837166478262528550	4
5	-0.735723910673131587	5	-0.735723910673131590	4
6	-0.614212712689667817	6	-0.614212712689667820	4
7	-0.475947393037073563	7	-0.475947393037073560	5
8	-0.324699469204683511	8	-0.324699469204683510	5
9	-0.164594590280733893	9	-0.164594590280733890	6
10	0.000000000000000000	10	0.000000000000000000	6
11	0.164594590280733893	10	0.164594590280733890	7
12	0.324699469204683511	11	0.324699469204683460	7
13	0.475947393037073563	11	0.475947393037073560	7
14	0.614212712689667817	12	0.614212712689667820	8
15	0.735723910673131587	12	0.735723910673131590	8
16	0.837166478262528546	11	0.837166478262528550	8
17	0.915773326655057396	10	0.915773326655057400	7
18	0.969400265939330374	8	0.969400265939330370	7
19	0.996584493006669847	8	0.996584493006669850	5

Table 3: Roots for $T_{19}(x)$ with both algorithms.

k	Analytical formula: $x_k = \cos\left(\frac{2k-1}{2N}\pi\right)$	MATLAB, x_k
1	-0.996584493006669847	-0.996584493006669850
2	0.969400265939330486	-0.969400265939330370
3	-0.915773326655057507	-0.915773326655057510
4	-0.837166478262528546	-0.837166478262528550
5	-0.735723910673131587	-0.735723910673131590
6	-0.614212712689667817	-0.614212712689667820
7	-0.475947393037073563	-0.475947393037073560
8	-0.324699469204683455	-0.324699469204683460
9	-0.164594590280733838	-0.164594590280734060
10	0.000000000000000061	0.000000000000000061
11	0.164594590280733977	0.164594590280733980
12	0.324699469204683566	0.324699469204683570
13	0.475947393037073618	0.475947393037073620
14	0.614212712689667817	0.614212712689667820
15	0.735723910673131698	0.735723910673131590
16	0.837166478262528657	0.837166478262528550
17	0.915773326655057396	0.915773326655057400
18	0.969400265939330374	0.969400265939330370
19	0.996584493006669847	0.996584493006669850

Table 4: Roots for $T_{19}(x)$ with analytical formula and MATLAB.

References

[1] M. Abramowitz and I. A. Stegun, Handbook of mathematical functions, Wiley and Sons, N. Y. (1972).

[2] C. Lanczos, Applied analysis, Dover N. Y. (1988).

[3] J. B. Seaborn, Hypergeometric functions and their applications, Springer-Verlag (1991).

[4] C. Lanczos, Linear differential operators, Dover N. Y. (1997).

[5] J. C. Mason and D. C. Handscomb, Chebyshev polynomials, Chapman & Hall - CRC Press (2002).

V. Barrera-Figueroa, J. Sosa-Pedroza, J. López-Bonilla
UPALM. Edif. Z-4, 3er. piso, col. Lindavista, C.P. 07738, México D.F.