A Fast Algorithm for the Calculation of the Roots of Special Functions

Zachary Murez

Abstract—I implemented an algorithm that finds the roots of functions, satisfying second-order differential equations, including the classical special functions. The algorithm is based on the paper “A Fast Algorithm for the Calculation of the Roots of Special Functions” by Glaser, Liu, and Rokhlin. The special functions have many applications including use in Gaussian Quadratures, resonances of mechanical and electrical systems, and quantum mechanical calculations. As such, algorithms for this task are well developed. Most run in time $O(n^2)$ where $n$ is the number of roots. Some newer algorithms achieve it in time $O(n\log n)$. This algorithm achieves it in time $O(n)$. It works in a two-step process. For each root an initial approximation is found by solving the differential equation given by the Prufer transform. This approximation is then improved using Newton’s method with evaluation of $f$ and $f’$ performed by a Taylor series approximation around the previous zero. The expansion is calculated by a recursive relation derived from the differential equation. I numerically tested the performance of the algorithm on the Legendre, and Jacobi polynomials.

I. Mathematical Tools

Ia. Second Order Differential Equation

The algorithm finds the zeros for any functions satisfying the following differential equation

$$p(x) \frac{d^2 u}{dx^2} + q(x) \frac{du}{dx} + r(x) u(x) = 0$$

where $p(x), q(x), r(x)$ are polynomials of degree 2.

Ib. Runge-Kutta method

The second-order Runge-Kutta method solves the initial-value problem

$$y(x_0) = y_0,$$

$$y'(x) = f(x, y)$$

on the interval $[x_0, x_0 + L]$ by taking a sequence of $n$ steps:

$$x_{i+1} = x_i + h,$$

$$k_{i+1} = hf(x_i + h, y_i + k_i),$$

$$y_{i+1} = y_i + \frac{1}{2} (k_i + k_{i+1})$$

where $h=L/n$ and $k_0=hf(x_0, y_0)$. This algorithm is of cost proportional to $n$, and has an error of order $h^2$.

Ic. Prufer Transform

The Prufer transform that is used is

$$\frac{dx}{d\theta} = -\left( \sqrt{\frac{r'}{p} + \frac{r'p - p'r + 2rpq}{2rp} \cdot \sin 2\theta} \right)^{-1},$$

where $0, p, q, r$ are functions of $x$. Now suppose $x_0$ is a root of $u$ and let $x$ be the function defined by the Prufer Transform with initial condition

$$x(\frac{\pi}{2}) = x_0$$

then for each root $\dot{x}$ of $u$, there exists an integer $n$ such that
thus, given one root of \( u \) all the roots can be found.

Id. Taylor Series

Given a sufficiently smooth function \( u \) and an initial point \( x_0 \) the values of \( u \) in a neighborhood of \( x_0 \) can be approximated by

\[
    u(x_0 + h) = \sum_{k=0}^{m} \frac{u^{(k)}(x_0)}{k!} h^k + \varepsilon
\]

And values of \( u' \) can be approximated by

\[
    u'(x_0 + h) = \sum_{k=1}^{m} \frac{u^{(k)}(x_0)}{(k-1)!} h^{k-1} + \varepsilon'
\]

In addition, the values of \( u^{(k)} \) can be evaluated by

\[
    pu^{(k+2)} = -(kp' + q)u^{(k+1)} - \left( \frac{k(k-1)}{2} p'' + kq' + r \right) u^{(k)} - \left( \frac{k(k-1)}{2} q'' + k^2 r' \right) u^{(k-1)} - \frac{k(k-1)}{2} r'' u^{(k-2)}
\]

Ie. Newton’s Method

Given an initial approximation \( x_0 \) Newton’s method solves the equation \( f(x)=0 \) iteratively by

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

Under normal conditions this converges quadratically.

If. Legendre Polynomials

The Legendre polynomials \( P_n(x) \) satisfy

\[
    (1 - x^2)y'' - 2x y' + n(n + 1)y = 0
\]

We note that \( P_n \) is symmetric. In addition, when \( n \) is odd, zero is a root. When \( n \) is even, zero is an extrema.

Ig. Jacobi Polynomials

The Jacobi polynomials \( J_n^{a,b}(x) \) satisfy

\[
    (1 - x^2)y'' + (b - a - (a + b + 2)x)y' + n(n + a + b + 1)y = 0
\]

II. The Algorithm

IIa. Main Algorithm

Input: the polynomial coefficients of \( p, q, r \), an initial root \( x_1 \), the derivative \( u'(x_1) \) and the number of roots \( N \) larger than \( x_1 \).

Output: an array of the \( N \) roots \( x_1, x_2, \ldots, x_N \) and an array of the derivatives \( u'(x_1), u'(x_2), \ldots, u'(x_N) \).

Set \( \text{roots}(1)=x_1 \) and \( \text{ders}(1)=u'(x_1) \).

Do \( i=1 \) to \( N-1 \)

1. Use Runge-Kutta to solve the Prüfer transform for \( \theta=\pi/2 \) to \(-\pi/2\) with \( x(\theta_0)=\text{roots}(i) \). Set \( \text{roots}(i+1) \) to \( x(-\pi/2) \).

2. Use the recurrence given to calculate the first \( m \) derivatives of \( u \) at \( x=\text{roots}(i) \), starting with \( u(x)=0, u'(x)=\text{ders}(i) \).

3. Use Newton’s method to improve the precision of \( \text{roots}(i+1) \) with \( u \) and \( u' \) evaluated by their Taylor expansions around \( \text{roots}(i) \). Set \( \text{ders}(i+1)=u'(\text{roots}(i+1)) \).

end do

IIb. Supplemental Algorithm

Input: the polynomial coefficients of \( p, q, r \), an initial extrema \( x_e \) of \( u \), the value of \( u(x_e) \)

Output: the smallest root \( x_1>x_e \) and the derivative \( u'(x_1) \).

1. Use Runge-Kutta to solve the Prüfer transform for \( \theta=0 \) to \(-\pi/2\) with \( x(\theta_0)=x_e \). Set \( x_1 \) to \( x(-\pi/2) \).

2. Use the recurrence given to calculate the first \( m \) derivatives of \( u \) at \( x=x_e \), starting with \( u(x)=x_e, u'(x)=0 \).

3. Use Newton’s method to improve the precision of \( x_1 \) with \( u \) and \( u' \) evaluated by their Taylor expansions around \( x_e \). Return \( x_1 \) and \( u'(x_1) \).

III. Implementation

The algorithms described in this paper were implemented in C. Calculations were performed using double precision variables. One difficulty was evaluating large factorials using integers; the results blew up, causing the results to only be accurate to a few decimal places. However, when these were changed to double precision computations, the problem went away. For the solution to the Prüfer transform ten Runge-Kutta steps were used. This gave an approximation good to two or three
decimals. Using more steps did not improve the accuracy and this was sufficient for a first approximation. As was done in the original paper, thirty terms of the Taylor series were used for the evaluation of \( u \) and \( u' \). The quadratic convergence of Newton’s method on the functions tested in this paper allowed for very rapid convergence to the final values. Seven iterations were sufficient to give results accurate to fifteen digits.

### IIIa. Legendre Polynomials

Since the Legendre polynomials are symmetric it is enough to calculate just the positive roots. The negative roots are given by the negatives of the positive roots. If the order is odd, 0 is a root. We give the value of \( u'(0) \) and use the main algorithm. If the order is even, 0 is an extrema. We use the supplemental algorithm to find the first root, and then the main algorithm to find the rest.

### IIIb. Jacobi Polynomials

The Jacobi polynomials are a generalization of the Legendre polynomials. In fact \( j_n^{0,0} = p_n \). However, when \( a \) and \( b \) are not zero, we get more complicated functions that are not necessarily symmetric, and don’t necessarily have a root or extrema at zero. When \( a=b \), the polynomials are symmetric. If it is of odd degree, zero is a root. If it is of even order zero is an extrema. If \( a\neq b \), the polynomials need not be symmetric and zero need not be a root or extrema. As such, when \( a=b \), we use the same procedures as in the case of the Legendre polynomials. When \( a\neq b \), the smallest root, and the value of the derivative there, is given and the main algorithm is used.

### IV. Results

Trials were run on an Intel Xeon X5550 quad-core processor with 12 Gigabytes of RAM, running Linux. Mean error was calculated using values given by Mathimatica. All numbers were recorded to 16 decimal places.

#### IVa. Legendre Polynomials

<table>
<thead>
<tr>
<th>Order</th>
<th>Roots</th>
<th>Mean Error</th>
<th>CPU Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>250</td>
<td>125</td>
<td>8E-17</td>
<td>10</td>
</tr>
<tr>
<td>500</td>
<td>250</td>
<td>8E-17</td>
<td>20</td>
</tr>
<tr>
<td>1000</td>
<td>500</td>
<td>2.3E-16</td>
<td>50</td>
</tr>
</tbody>
</table>

#### IVb. Jacobi Polynomials

<table>
<thead>
<tr>
<th>Order</th>
<th>Roots</th>
<th>Mean Error</th>
<th>CPU Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>250</td>
<td>125</td>
<td>8E-17</td>
<td>10</td>
</tr>
<tr>
<td>500</td>
<td>250</td>
<td>8E-17</td>
<td>20</td>
</tr>
<tr>
<td>1000</td>
<td>500</td>
<td>2.3E-16</td>
<td>50</td>
</tr>
</tbody>
</table>

#### a=70, b=70

<table>
<thead>
<tr>
<th>Order</th>
<th>Roots</th>
<th>CPU Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>250</td>
<td>125</td>
<td>10</td>
</tr>
<tr>
<td>500</td>
<td>250</td>
<td>20</td>
</tr>
<tr>
<td>1000</td>
<td>500</td>
<td>50</td>
</tr>
<tr>
<td>2000</td>
<td>1000</td>
<td>100</td>
</tr>
<tr>
<td>4000</td>
<td>2000</td>
<td>200</td>
</tr>
<tr>
<td>8000</td>
<td>4000</td>
<td>420</td>
</tr>
<tr>
<td>16000</td>
<td>8000</td>
<td>840</td>
</tr>
<tr>
<td>32000</td>
<td>16000</td>
<td>1680</td>
</tr>
</tbody>
</table>

#### a=12, b=19

<table>
<thead>
<tr>
<th>Order</th>
<th>Roots</th>
<th>Mean Error</th>
<th>CPU Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>250</td>
<td>250</td>
<td>1.3E-15</td>
<td>20</td>
</tr>
</tbody>
</table>

### V. Conclusions

From the numerical results, we confirm that the algorithm runs in linear time in the number of roots found. Interestingly, we also find that the CPU time is largely unaffected by the more complicated Jacobi polynomials when compared to the simpler Legendre polynomials. The original paper tested the algorithm on many more diverse special functions and also found that the speed was largely independent of the class of function. The mean error per root was at the machine precision for double precision values. Most values were exact to 16 digits and the ones that were off were off by one in the last digit. Many tests for accuracy were unable to be performed due to lack of theoretical values. Mathimatica is to slow to be feasible when looking for more than 500 roots. Future work could involve testing on extended precision values, as was done in the original paper.