Hardy Z Software Program

4.1 Introduction

The contents of this section is more fully discussed in my book, *A Study of Riemann's Zeta Function*, Terrence P. Murphy.

The Hardy Z function, $Z(t)$, greatly simplifies the location of zeros of $\zeta(s)$ (the zeta function) on the critical line ($Re(s) = \frac{1}{2}$). We have

$$Z(t) = \zeta\left(\frac{1}{2} + it\right) e^{i\theta(t)},$$

where $\theta(t)$ is given by the estimate

$$\theta(t) \approx \frac{t}{2}\log\frac{t}{2\pi} - \frac{\pi}{8} - \frac{t}{2} + \frac{1}{48t} + \frac{7}{5760t^3}.$$

The error in the approximation of $\theta(t)$ is less than a very small fraction of the t^{-3} term and therefore approaches zero very rapidly for increasing t.

For t real, $|Z(t)| = \left|\zeta\left(\frac{1}{2} + it\right)\right|$. Thus, $Z(t) = 0$ if and only if $\zeta\left(\frac{1}{2} + it\right) = 0$. Also, for t real, $Z(t)$ is real. With $Z(t)$ a smooth function along the critical line, that means that $Z(t) = 0$ only if $Z(t)$ changes sign in the immediate neighborhood of that zero.

Thus, the task of finding zeros of the zeta function on the critical line is reduced to finding sign changes of Hardy's Z function. In the simplest case, we know that one or more zeta zeros must exist *somewhere* in an interval on the critical line if $Z(t)$ has different signs at the interval end points.

More often, the goal is to find zeta zeros with great accuracy. For that, we must estimate $Z(t)$ efficiently and with a minimal error term. The Riemann-Siegel Formula provides a remarkably accurate and efficient method of computing $Z(t)$. For a given t, the Riemann-Siegel Formula requires just $\mathcal{O}\left(t^{1/2}\right)$ computation steps. We describe that formula next.

Fix any $t > 0$ and let $T = (t/2\pi)^{1/2}$. Let $N = [T]$ and $p = \{T\}$ be the integral and fractional parts of T. Then, for the "main term" $M(t)$

$$M(t) = 2\sum_{n=1}^{N} n^{-1/2}\cos\left[\theta(t) - t\log n\right],$$

we have

$$Z(t) = M(t) + R,$$

where R is called the "remainder term". The Riemann Siegel Formula gives the following estimate:

$$R \approx R(K) = (-1)^{N-1}\left(\frac{t}{2\pi}\right)^{-1/4}\left[\sum_{j=0}^{K} C_j\left(\frac{t}{2\pi}\right)^{-j/2}\right] \quad (K \in \mathbb{N}).$$

For $K = 4$, the five C_j terms are

$$C_0 = \Psi(p) = \frac{\cos\left[2\pi\left(p^2 - p - 1/16\right)\right]}{\cos(2\pi p)}$$

$$C_1 = -\frac{\Psi^{(3)}(p)}{96\pi^2}$$

$$C_2 = \frac{\Psi^{(6)}(p)}{18,432\pi^4} + \frac{\Psi^{(2)}(p)}{64\pi^2}$$

$$C_3 = -\frac{\Psi^{(9)}(p)}{5,308,416\pi^6} - \frac{\Psi^{(5)}(p)}{3,840\pi^4} - \frac{\Psi^{(1)}(p)}{64\pi^2}$$

$$C_4 = \frac{\Psi^{(12)}(p)}{2,038,431,744\pi^8} + \frac{11\cdot\Psi^{(8)}(p)}{5,898,240\pi^6} - \frac{19\cdot\Psi^{(4)}(p)}{24,676\pi^4} + \frac{\Psi(p)}{128\pi^2}.$$

It turns out the C_j terms are entire functions that can be expressed as power series. In our program, we use a table of power series coefficients developed by Haselgrove to compute the approximate remainder term $R(4)$. Considering only the mathematical theory discussed above, the error in the approximation of $Z(t)$ given by $Z(t) = M(t) + R(4)$ is $\leq \left|0.017t^{-11/4}\right|$ for $t > 200$.

Of course, the mathematical theory above does not account for the errors introduced by the real world of floating point calculations. For N as defined above, $M(t)$ requires N floating point calculations of square root, cosine and logarithm, with associated sums and products. For larger t (and therefore larger N), this requires a large number of quite accurate floating point calculations. We discuss further below the C compiler and specialized floating point library used in our program.

With our current compiler and environment, the results seem to be accurate to 5 decimal digits for $t \leq 200$ and to at least 6 decimal digits for $t > 200$. In fact, as t increases the accuracy increases. For example, for $t = 306,100,245,898.0967563914567902$ (the $103,800,788,000^{\text{th}}$ zeta zero), the program is accurate to 14 decimal digits.

4.2 Compiler Setup on Windows

We are using the gcc/mingw-64 C compiler on Windows 11, with the MPFR library – a C library for multiple-precision floating-point computations, with user-selectable precision and correct rounding. Our program runs with default floating point precision set to 256 bits (typically, a **double** floating point type in C has 64 bits).

We briefly discuss here how we installed the **gcc** compiler and the MPFR floating point library on our Windows system. If you do not have a Windows 10 or 11 system, a different setup will be needed. If you Google **gcc C compiler** and **MPFR library**, along with your computer system, you should find installation instructions.

The following is to the best of my recollection. I make no warranties on any of the steps, methods or programs described below. Proceed at your own risk.

(1) Download the GUI installer from `www.msys2.org`.

(2) Run the installer. The GUI will ask for an install folder. The default of `c:\msys64` should be accepted.

(3) When done, click Finish on the GUI.

(4) A Unix-style terminal will now launch – all installations are done inside that terminal. Install the **gcc** compiler and a pre-selected group of supporting libraries (including MPFR) using the **pacman** installer program as follows.

```
pacman -S mingw-w64-ucrt-x86_64-toolchain
```

(5) After the needed files are downloaded, respond 'Y' to "Proceed with installation".

(6) When completed, exit the terminal (click the 'X' in the upper-right corner of the terminal, or type exit).

(7) Next, you need to add `c:\msys64\ucrt64\bin` to your Windows (user) PATH. To do that, just Google how to add a directory to your PATH.

(8) Now verify **gcc** was installed by typing the following at the Windows command prompt:

```
gcc --version
```

(9) If a **gcc** version number is displayed (mine was 14.1.0), the installation is complete.

4.3 Making the Hardy Z Program on Windows

Once the compiler installation is complete, you can make the `HardyZ.exe` program.

(1) Create a new directory `gcc` under your Documents directory. Then create a new directory `HardyZ` under the `gcc` directory. You then have: `Documents\gcc\HardyZ`.

(2) Copy `hardyZ.c, hardyZ.h, computeMain.c, remainder.c, remainder128.c` and `makehardyZ.bat` to the newly created `HardyZ` directory.

(3) Open the Command prompt, change directory to `Documents\gcc\HardyZ` and run the `makehardyZ.bat` batch file. The `hardyZ.exe` program will be ready for use.

4.4 Using the Hardy Z Program

Our program is a command line program that takes the following command line parameters:

-t [positive number]	The t value for Z(t) - this parameter is required. (Digits and '.' only).
-i [positive number]	Amount to increment t (if checking multiple t values) – defaults to 1.
-c [positive integer]	Count of the number of t values to check – defaults to 1.
-z [positive integer]	Decimal points of Z(t) to show in report – defaults to 6.
-b [positive integer]	Floating point bits - $128 \leq b \leq 1024$, divisible by 64 – defaults to 256.
-h	Show command line parameters. All other parameters will be ignored.
-s	Report the total seconds taken to compute the Hardy Z values.
-v	Verbose report (otherwise CSV only).

As an example, entering the 80^{th} zeta zero on the command line:

```
hardyz -t 201.26475194370378873301613342754 82
```

should give the following output

```
201.26475194370378873301613342754 82, -0.000000
```

The output is in CSV format, with t followed by $Z(t)$. Entering

```
hardyz -t 201.26475 -c 2 -i 0.00001
```

should give the following output

```
201.26475, 0.000007
201.26476, -0.000027
```

The sign change shows there is a zeta zero between 201.26475 and 201.26476.

4.5 Checking the Accuracy of Hardy Z Values

To check the accuracy of output from the `HardyZ.exe` program, I use the web site:

```
https://www.lmfdb.org/zeros/zeta/
```

That site has a database containing the first 103,800,788,359 zeros of the Riemann zeta function that have: (1) real part equal to $\frac{1}{2}$, and (2) imaginary part greater than zero. The imaginary parts have been computed to a precision of $\pm 2.5 \times 10^{-31}$.

4.6 The Source Code for the Hardy Z Program

4.6.1 hardyZ.c

This source code file is quite straightforward. We validate the user's command line input, save their choices and call the `ComputeHardyZ` function.

4.6.2 remainder128.c

See the discussion above about the "remainder" term R in computing Hardy Z values. In our code, we call the `ComputeRemainder128` function to compute the $R(4)$ approximation of R. We use the Haselgrove table of power series coefficients for the C_0 through C_4 terms.

In this source code file, we use the **gcc** `quadmath` 128-bit floating point library for the Haselgrove coefficients, and the MPFR library (set to 256-bit by default and changeable by the "-b" command line parameter) for all other floating point calculations.

4.6.3 remainder.c

Here we provide an alternate function, `ComputeRemainder`, for computing the "remainder" term. Like the `ComputeRemainder128` function, we compute $R(4)$ using the Haselgrove table of power

series coefficients. The difference is that, here, we use the **gcc** compiler's built-in **long double** floating point type (with 80-bit precision on our computer system).

Our testing showed no meaningful increase in accuracy when using `ComputeRemainder128`. However, the time cost of using `ComputeRemainder128` was also very minimal. For that reason, we default to using `ComputeRemainder128`. To test using `ComputeRemainder` instead, use the "-d" command line parameter, setting the value to any positive integer ≤ 210 that is divisible by 3.

4.6.4 computeMain.c

This source code file is"central control" for computing the Hardy Z values. The `ComputeHardyZ` function is the entry point. In that function, we

(1) Initialize the MPFR floating point system.

(2) Begin the loop computing Hardy Z values. If no "-c" command line parameter is entered, the loop ends after the first Hardy Z computation. Otherwise we loop the number of times requested in the "-c" command line parameter. After each loop iteration, we increment t by either 1 (the default value) or the amount requested in the "-i" command line parameter.

(3) Inside the loop, we: (1) compute the N and p variables of the Riemann-Siegel Formula, (2) call the `ComputeRemainder128` function to compute the $R(4)$ approximated remainder term, (3) call the `ComputeMain` function to compute the main term, and (4) print the results to **stdout**.

(4) After exiting the loop, we close down the MPFR system.

Also in this source code file is the `ComputeMain` function. This is where we compute the "main" term of the Riemann-Siegel formula. In that function, we

(1) Using the passed parameter t, compute $\theta(t)$.

(2) Using the passed parameter N, loop $n = 1$ to $n = N$ times, compute $n^{-1/2}\cos\left[\theta(t) - t\log n\right]$ in each loop and sum those computed values in a `Main` variable.

(3) At the end of the loop, multiply `Main` by 2 and return that result.

For the nth argument of cosine in the above formula, let $A(n) = \theta(t) - t\log n$. There are multiple choices for calculating $A(n)$. We use the following idea in our approach

$$\begin{aligned} A(n) - A(n-1) &= \left[\theta(t) - t\log n\right] - \left[\theta(t) - t\log(n-1)\right] \\ &= t\log(n-1) - t\log n = t\left[\log(n-1) - \log(n)\right] \\ A(n) &= A(n-1) + t\left[\log(n-1) - \log(n)\right]. \end{aligned}$$

Note that $\left[\log(n-1) - \log(n)\right]$ is a very small negative number that gets smaller as n gets larger. When multiplied by t, the product is therefore much smaller than t. The alternative, computing $t\log n$ (for larger n) will produce a product much larger than t. Our approach keep an intermediate variable smaller that t, which *may* have accuracy benefits in our floating point calculations. To test the standard calculation method (i.e., computing $t\log n$), use the "-d" command line parameter, setting the value to any positive integer ≤ 210 that is divisible by 7.

We keep intermediate variables smaller in one other way. Before passing $A(n)$ to the cosine function, we use the `mpfr_fmod` function to set $A(n)$ to the remainder obtained by dividing $A(n)$ by 2π. This is justified because the cosine function is 2π periodic. No doubt, the `mpfr_cos` function will otherwise do the same thing. To bypass using the `mpfr_fmod` function, use the "-d" command line parameter, setting the value to any positive integer ≤ 210 that is divisible by 5.

4.7 Performance of the Hardy Z Program

To give a general idea of the performance of the Hardy Z program, I will list the time it takes on my computer to do several different calculations. My computer is a Windows 11 computer, with an Intel Core i7-14700 (2.10 Ghz) processor and 32.0 GB of memory.

t	Count	Increment	MPFR bits	Time (Seconds)
1,234.5678	10	0.10	256	0.005
1,234.5678	100	0.10	256	0.041
9,871,234.5678	10	0.10	256	0.088
9,871,234.5678	10	0.10	**512**	0.132
9,871,234.5678	100	0.10	256	0.926
6,789,871,234.5678	10	0.10	256	2.174
6,789,871,234.5678	10	0.10	**512**	3.167
6,789,871,234.5678	100	0.10	256	21.76
416,789,871,234.5678	10	0.10	256	17.59

Consider the last t number above. A single calculation of the "main" term (i.e., Count = 1) involves N iterations through a loop computing $n^{-1/2}\cos\left[\theta(t) - t\log n\right]$ where $N = 256,554$.

4.8 Other Notes

I should mention that the **gcc** compiler is licensed from Free Software Foundation, Inc. The license is described at:

`https://gcc.gnu.org/onlinedocs/libstdc++/manual/license.html`

The (also generous) license for the MPFR floating point software is shown at:

`https://www.mpfr.org/`