C Library for Hardy Z Values, Gram Points and Turing's Method

1.1 Introduction

The `libhgt.a` file is a library of functions that support: (1) locating zeros of $\zeta(s)$ (the zeta function) on the critical line, and (2) up to a given height T, verifying that there are no zeros **off** the critical line. Specifically, the functions allow you to compute Hardy Z values, locate Gram Points, and calculate all values needed to apply Turing's Method. The library also provides functions to validate user input. This ensures that all calls to the calculating functions include only well-formed input.

In this paper, we will do as follows:

- Provide the syntax, parameters and return value for all public-facing library functions.
- Provide a high-level description of the Hardy Z function and its association with zeros of the zeta function. This includes the method used (the Riemann-Siegel formula) to calculate Hardy Z values.
- Provide a high-level description of Gram Points. We discuss how they are calculated and why they are important in locating zeros of $\zeta(s)$.
- Provide a high-level description of Turing's Method and explain how it is used to verify the Riemann Hypothesis up to a given T on the critical line.
- Describe the compiler environment, including the third-party libraries used. This includes the MPFR library – a C library for multiple-precision floating-point computations, with user-selectable precision and correct rounding.
- Discuss the timed performance of the calculation-intensive Hardy Z function, particularly in the case of large values of T on the critical line.

1.2 Public Facing Functions

The public facing library functions are in four groups: (1) functions to process and validate user input, (2) functions to initialize and close the MPFR floating point system, (3) functions to calculate Hardy Z values, and (4) functions to locate Gram Points.

In that order, we will discuss the function names, syntax, parameters and return values.

1.2.1 Functions to Process / Validate User Input

We assume that any program using the `libhgt.a` library of functions will first obtain parameter data from either the command line or other user input mechanism. Before calling the Hardy Z or Gram

Point library functions, that parameter data should be processed and validated through the following group of library functions.

Function: ValidateHardyT

Syntax

```
int ValidateT(const char *str)
```

Parameters

- ***str*** – *This is the string to validate. The string must be a well-formed positive number (digits and at most one period), representing the t of the point ($\frac{1}{2} + it$) on the critical line. Valid values are between `HGT_HARDY_T_MIN` and `HGT_HARDY_T_MAX` (see `hgt.h`).*

Return Value

Returns 1 if successful, and a negative number otherwise.

Function: ValidateIncr

Syntax

```
int ValidateIncr(const char *str)
```

Parameters

- ***str*** – *This is the string to validate. The string must be a well-formed positive number (digits and at most one period), representing the amount to increment an initially provided t of ($\frac{1}{2} + it$). This is used when the function `HardyZWithCount` is called to process multiple t's. Valid values are between `HGT_T_INCR_MIN` and `HGT_T_INCR_MAX` (see `hgt.h`).*

Return Value

Returns 1 if successful, and a negative number otherwise.

Function: ValidateCount

Syntax

```
int ValidateCount(const char *str)
```

Parameters

- ***str*** – *This is the string to validate. The string must be a well-formed whole positive number (digits only), representing the count of the number of times some action is performed. Valid values are between `HGT_COUNT_MIN` and `HGT_COUNT_MAX` (see `hgt.h`).*

Return Value

Returns the count value if successful, and −1 otherwise.

Function: ValidateThreads

Syntax

```
int ValidateThreads(const char *str)
```

Parameters

- ***str*** – *This is the string to validate. The string must be a well-formed whole positive number (digits only), representing the number of compute threads to use in calculations. Valid values are between `HGT_THREADS_MIN` and `HGT_THREADS_MAX` (see `hgt.h`).*

Return Value

Returns the number of threads if successful, and −1 otherwise.

Function: ValidateDebugFlags

Syntax

```
int ValidateDebugFlags(const char *str)
```

Parameters

- ***str*** – *This is the string to validate. The string must be a well-formed whole positive number (digits only), representing the debug flags to set. A particular debug flag is set if the positive number is evenly divisible by the prime number representing that debug flag. Valid values are between `HGT_DEBUG_MIN` and `HGT_DEBUG_MAX` (see `hgt.h`).*

Return Value

Returns the a positive number representing debug flags if successful, and −1 otherwise.

Function: ValidatePrecisionMPFR

Syntax

```
int ValidatePrecisionMPFR(const char *str)
```

Parameters

- ***str*** – *This is the string to validate. The string must be a well-formed whole positive number (digits only), representing the number of precision bits to use in calls to the MPFR library. Valid values are between `HGT_PRECISION_MIN` and `HGT_PRECISION_MAX` (see `hgt.h`).*

Return Value

Returns the number of precision bits if successful, and -1 otherwise.

Function: ValidateReportDecimalPlaces

Syntax

```
int ValidateReportDecimalPlaces(const char *str)
```

Parameters

- ***str*** – *This is the string to validate. The string must be a well-formed whole positive number (digits only), representing the number of decimal places to use in certain output reports. Valid values are between `HGT_RPT_DEC_PLACES_MIN` and `HGT_RPT_DEC_PLACES_MAX` (see `hgt.h`).*

Return Value

Returns the number of decimal places (possibly zero) if successful, and -1 otherwise.

Function: GetDecimalDigits

Syntax

```
int GetDecimalDigits(const char *str)
```

Parameters

- ***str*** – *It is assumed that this string has already been validated as a number. Here, we count the number of decimal digits in the string.*

Return Value

Returns the number of decimal digits in the passed string.

Function: ValidateGramT

Syntax

```
int ValidateGramT(const char *str)
```

Parameters

- ***str*** *– This is the string to validate. The string must be a well-formed positive number (digits and at most one period), representing the t of the point* $(\frac{1}{2} + it)$ *on the critical line. Valid values are between* `HGT_GRAM_T_MIN` *and* `HGT_GRAM_T_MAX` *(see* `hgt.h`*).*

Return Value

Returns 1 if successful, and a negative number otherwise.

Function: ValidateGramN

Syntax

```
int ValidateGramT(const char *str)
```

Parameters

- ***str*** *– This is the string to validate. The string must be a well-formed integer greater than* -1 *(digits and at most one minus sign), representing the Nth Gram Point to calculate. Valid values are between* `HGT_GRAM_N_MIN` *and* `HGT_GRAM_N_MAX` *(see* `hgt.h`*).*

Return Value

Returns 1 if successful, and a negative number otherwise.

Function: ValidateGramAccuracy

Syntax

```
int ValidateGramAccuracy(const char *str)
```

Parameters

- ***str*** – *This is the string to validate. The string must be a well-formed whole positive number (digits only), representing the number of decimal places of accuracy required in calculating a Gram Point. Valid values are between `HGT_GRAM_ACCURACY_MIN` and `HGT_GRAM_ACCURACY_MAX` (see `hgt.h`).*

Return Value

Returns the number of decimal places if successful, and −1 otherwise.

Function: ValidateTuringGramPoints

Syntax

```
int ValidateTuringGramPoints(const char *str)
```

Parameters

- ***str*** – *This is the string to validate. The string must be a well-formed whole positive number (digits only), representing the number of Gram Points to use in the Turing Method calculations. Valid values are between `HGT_TUR_GRAM_PTS_MIN` and `HGT_TUR_GRAM_PTS_MAX` (see `hgt.h`).*

Return Value

Returns the number of decimal places if successful, and −1 otherwise.

Function: ValidateTuringSubIntervals

Syntax

```
int ValidateTuringSubIntervals(const char *str)
```

Parameters

- ***str*** – *This is the string to validate. The string must be a well-formed whole positive number (digits only), representing the number of subintervals to use between Gram Points in the Turing Method calculations. Valid values are between `HGT_TUR_SUBINTVL_MIN` and `HGT_TUR_SUBINTVL_MAX` (see `hgt.h`).*

Return Value

Returns the number of decimal places if successful, and −1 otherwise.

1.2.2 Functions to Initialize and Close the MPFR System

The `InitMPFR` function must be called before first use of the MPFR floating point system. After the last use of the MPFR system, the `CloseMPFR` function must be called.

Function: InitMPFR

Syntax

```
int InitMPFR(int DefaultBits, int MaxThreads, int DebugFlags, bool
CalcHardy)
```

Parameters

- ***DefaultBits*** *– This is the number of precision bits to use in calls to the MPFR library.*
- ***MaxThreads*** *– This is the maximum number of threads to use in calculations.*
- ***DebugFlags*** *– This is a debug number that may be used during internal testing.*
- ***CalcHardy*** *– TRUE if Hardy Z values will be calculated, FALSE otherwise.*

Return Value

Returns 1.

Function: CloseMPFR

Syntax

```
int CloseMPFR(void)
```

Parameters

- ***NONE.***

Return Value

Returns 1.

1.2.3 Functions to Calculate Hardy Z Values

The `HardyZWithCount` function is the most important function in the `libhgt` library. Its primary purpose is to compute the Hardy Z value at $(\frac{1}{2} + it)$, for a given positive real number t. It can also be used to compute multiple Hardy Z values, beginning at t and including equal-spaced increases in the value of t. The computed Hardy Z value(s) are returned through a user-provided callback function (which was passed as a parameter to `HardyZWithCount`).

Several non-public functions of `libhgt` are used to calculate Hardy Z values. Those functions use the Riemann-Siegel formula. For more information on those functions and an overview of the Riemann-Siegel formula, see section 1.4.

Function: HardyZWithCount

Syntax

```
int HardyZWithCount(mpfr_t t, mpfr_t Incr, int Count, int CallerID,
                          pHardyZCallback pCallbackHZ)
```

Parameters

- ***t*** *– For the given t, we are requesting the Hardy Z value at* $(\frac{1}{2}+it)$*.*
- ***Incr*** *– If the* **Count** *parameter is greater than* 1*, we increment t by* ***Incr*** *in each successive calculation of the Hardy Z value.*
- ***Count*** *– You are requesting* **Count** *calculations of the Hardy Z value; the first at t, the second at t+* ***Incr****, etc.*
- ***CallerID*** *– For the callback function you provide in the last parameter, this* ***CallerID*** *value is passed back to you. This may be needed if you call the* `HardyZWithCount` *function more than once.*
- ***pCallbackHZ*** *– This is a pointer to a function provided by you. The* `HardyZWithCount` *function calls this function with the results of its calculation of the Hardy Z value. This technique is needed to support the use of threading in the calculation of multiple Hardy Z values (i.e., Count > 1).*

Return Value

Returns 1.

Function: HardyZCallback

Syntax

```
int HardyZCallback(mpfr_t t, mpfr_t HardyZ, int i, int CallerID)
```

Parameters

- ***t*** *– This is the t (or the* ***Incr*** *of that t) passed to the* `HardyZWithCount` *function.*
- ***HardyZ*** *– This is the calculated Hardy Z value of the given t (or the* ***Incr*** *of that t).*
- ***i*** *– This is the zero-based index to the current* **Count** *number. Assume you passed* ***t = 1000*** *and* ***Count = 3*** *and* ***Incr = 0.25*** *to* `HardyZWithCount`*. If* ***i = 0*** *then the passed Hardy Z value is for* ***t = 1000****. If* ***i = 1*** *then the passed Hardy Z value is for* ***t = 1000.25****. If* ***i = 2*** *then the passed Hardy*

*Z value is for **t = 1000.50**.*

- ***CallerID** – This is the **CallerID** value you passed to `HardyZWithCount`. This may be needed if you call the `HardyZWithCount` function more than once.*

Return Value

Returns 1.

1.3 Functions to Calculate Gram Points

As discussed in section 1.5, Gram Points are an important component of locating zeros on the critical line and (along with Turing's Method) verifying the Riemann Hypothesis up to a given *T*.

The `GramAtN` function locates the *N*th Gram Point; or, optionally, locates a series of Gram Points beginning with the given *N*. The `GramNearT` function locates the largest Gram Point with a value less than or equal to the given *T*.

Function: GramAtN

Syntax

```
int GramAtN(mpfr_t *Result, mpfr_t N, mpfr_t Accuracy)
```

Parameters

- ***Result** – This is a pointer to an MPFR variable which will hold the result of the Gram Point calculation.*
- ***N** – The function is asked to calculate the Nth Gram Point.*
- ***Accuracy** – Accuracy is a (very) small number indicating the required accuracy of the calculation. For example, you might require accuracy to ten decimal digits.*

Return Value

Returns 1.

Function: GramNearT

Syntax

```
int GramNearT(mpfr_t *Result, mpfr_t T, mpfr_t Accuracy)
```

Parameters

- ***Result*** *– This is a pointer to an MPFR variable which will hold the result of the Gram Point calculation. We are looking for the largest Gram Point that is ≤ T.*
- ***T*** *– The function is asked to calculate the largest Gram Point that is ≤ T.*
- ***Accuracy*** *– Accuracy is a (very) small number indicating the required accuracy of the calculation. For example, you might require accuracy to ten decimal digits.*

Return Value

Returns 1.

1.4 The Hardy Z Function

The contents of this section are 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 the Hardy 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

$$\begin{aligned}
C_0 &= \Psi(p) = \frac{\cos\left[2\pi\left(p^2 - p - 1/16\right)\right]}{\cos(2\pi p)}\\
C_1 &= -\frac{\Psi^{(3)}(p)}{96\pi^2}\\
C_2 &= \frac{\Psi^{(6)}(p)}{18,432\pi^4} + \frac{\Psi^{(2)}(p)}{64\pi^2}\\
C_3 &= -\frac{\Psi^{(9)}(p)}{5,308,416\pi^6} - \frac{\Psi^{(5)}(p)}{3,840\pi^4} - \frac{\Psi^{(1)}(p)}{64\pi^2}\\
C_4 &= \frac{\Psi^{(12)}(p)}{2,038,431,744\pi^8} + \frac{11\cdot\Psi^{(8)}(p)}{5,898,240\pi^6} - \frac{19\cdot\Psi^{(4)}(p)}{24,676\pi^4} + \frac{\Psi(p)}{128\pi^2}.
\end{aligned}$$

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 Gabcke (accurate to 50 decimal places). With the Gabcke coefficients, we can compute the remainder term $R(4)$ with great accuracy. 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 $t > 200$.

The C code in `libhgt.a` to compute the Riemann-Seigel Formula for a given t can be found in these files: `RSbuildcoeff.c` (where we build the Gabcke coefficients), `RSmainTerm.c` (where we compute the main term), `RSremainder.c` (where we compute the remainder term) and `ThetaOfT.c` (where we compute the value of Theta).

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 $t \leq 200$, to at least 6 decimal digits for $t > 200$, to at least 10 decimal digits for $t > 1000$, to at least 12 decimal digits for $t > 10{,}000$, and to at least 28 decimal digits for $t > 10{,}000{,}000{,}000$. As you see, as t increases the accuracy increases.

1.5 Gram Points

We use here $\theta(t)$ as defined above for the Hardy Z function. For integers $n \geq -1$, we calculate Gram Point g_n by solving the equation $\theta(g_n) = \pi n$.

A simple calculation shows that $\theta'(t) \approx \frac{1}{2}\log\left(\frac{t}{2\pi}\right)$ and thus $\theta'(t) > 0$ for $t \geq 7$. It follows that $\theta(t)$ is *strictly increasing* for $t \geq 7$.

The first two Gram Points, g_{-1} and g_0, have approximate values 9.666908056 and 17.8455995404, respectively. We consider here only Gram Points for $n \geq 1$. In that way, we can be sure that $\theta(t)$ is strictly increasing between any two Gram Points.

Now consider $Z(t)$, the Hardy Z function, where t is a Gram Point. Recall from above, we have for $t \in \mathbb{R}$

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

and therefore

$$\begin{aligned}\zeta\left(\frac{1}{2} + it\right) &= e^{-i\theta(t)} Z(t) \\ &= Z(t)\cos\left[\theta(t)\right] - iZ(t)\sin\left[\theta(t)\right].\end{aligned}$$

Thus, along the critical line at Gram Point $t = g_n$, we have $\theta(g_n) = n\pi$, so that

$$\begin{aligned}Re\left[\zeta\left(\frac{1}{2} + ig_n\right)\right] &= Z(g_n)\cos\left[\theta(g_n)\right] = (-1)^n Z(g_n) \\ Im\left[\zeta\left(\frac{1}{2} + ig_n\right)\right] &= -Z(g_n)\sin\left[\theta(g_n)\right].\end{aligned}$$

Next consider the interval along the critical line where $t \in I_n = [g_n, g_{n+1}]$. In the *interior* of I_n, note that $\sin[\theta(t)]$ is never zero and does not change sign. Thus, a zero of $Im\left[\zeta\left(\frac{1}{2} + it\right)\right]$ is a zero of $Z(t)$ and therefore a zero of $\zeta\left(\frac{1}{2} + it\right)$. Put differently, any change of sign of $Im\left[\zeta\left(\frac{1}{2} + it\right)\right]$ in the *interior* of I_n is a change of sign of $Z(t)$ and therefore a zero of $\zeta\left(\frac{1}{2} + it\right)$.

By the definition of g_n, we have $\sin[\theta(g_n)] = \sin[\theta(g_{n+1})] = 0$. Therefore, *on the boundaries* of I_n, we have $Im\left[\zeta\left(\frac{1}{2} + ig_n\right)\right] = Im\left[\zeta\left(\frac{1}{2} + ig_{n+1}\right)\right] = 0$. Now consider a plot of the values of $\zeta(s)$ along the critical line $s = \frac{1}{2} + it$. The imaginary part will be zero if and only if either $\left(\frac{1}{2} + it\right)$ is a zero of $\zeta(s)$ or t is a Gram Point.

Continuing with I_n, we now turn to the real part of $\zeta\left(\frac{1}{2} + it\right)$. At Gram Points, empirical evidence suggests that $Re\left[\zeta\left(\frac{1}{2} + ig_n\right)\right]$ is usually positive. When that is true for both g_n and g_{n+1}, we have $(-1)^n Z(g_n) > 0$ and $(-1)^{n+1} Z(g_{n+1}) > 0$. In that case, $Z(g_n)$ and $Z(g_{n+1})$ have opposite signs and we therefore must have $Z(t) = 0$ for some t in the interior of I_n. We actually know even more. With opposite signs, there must be an *odd* number of zeta zeros in the interval; with the same sign, there must be an *even* number of zeta zeros in the interval.

It is apparent that the use of Gram Points greatly facilitates the location of zeta zeros.

1.6 Turing's Method

Define the rectangle $R(T) = \{s \in \mathbb{C} : 0 \le Re(s) \le 1, 0 \le Im(s) \le T\}$. The function $N(T)$ calculates the number of zeros of $\zeta(s)$ in $R(T)$.

Now suppose, using the Riemann-Siegel formula, you have located all 10,142 zeros on the critical line with imaginary parts between 0 and $T = 10{,}000$. To confirm the Riemann Hypothesis is true up to that point, you need to show $N(T) = 10{,}142$. That is, you need to show there are no zeros **off** the critical line.

It turns out that accurate calculations of $N(T)$ are far from simple. With $\theta(T)$ as defined above, and with $S(T)$ as discussed below, we have (assuming T is not the ordinate of a zero of $\zeta(s)$):

$$N(T) = \frac{1}{\pi}\theta(T) + 1 + S(T). \tag{1.1}$$

For the contour γ that runs from $(2+i0) \to (2+iT) \to (\frac{1}{2}+iT)$, we have

$$\begin{aligned} S(T) &= \frac{1}{\pi} Im \left(\int_\gamma \frac{\zeta'(s)}{\zeta(s)}\, ds \right) = \frac{1}{\pi} Im \left(\int_\gamma \frac{d}{ds} \log(\zeta(s))\, ds \right) \\ &= \frac{1}{\pi} Im \left(\int_\gamma \frac{d}{ds} \left[\log(|\zeta(s)|) + i \arg \zeta(s) \right] ds \right) \\ &= \pi^{-1} \triangle_\gamma \arg \zeta(s). \end{aligned}$$

where the change in argument of $\zeta(s)$ along γ is calculated using the continuous argument function.

The difficulty in calculating $S(T)$ causes the difficulty in calculating $N(T)$. An important advance in understanding $S(T)$ was made by Alan Turing. Given the 3-tuple (A, B, t_0), Turing's Theorem states that for $A = 2.065$, $B = 0.128$ and $t_0 = 168\pi$

$$\left| \int_{t_1}^{t_2} S(t)\, dt \right| \le A + B \log t_2, \quad \text{for } t_2 > t_1 > t_0.$$

It was later shown by Trudgian that the 3-tuple $(2.067, 0.059, 168\pi)$ satisfies the theorem.

Also, showing the usefulness of Gram Points, at Gram Point g_n, we have $\pi^{-1}\theta(g_n) = n.$ and thus

$$\begin{aligned} N(g_n) &= n + 1 + S(g_n) \\ S(g_n) &= N(g_n) - (n+1). \end{aligned}$$

With $N(g_n)$ an integer, $S(g_n)$ must also be an integer at all Gram Points. The mathematics is too complex to show here, but, in many circumstances the following is true. If $|S(T)| < 2$ at a Gram Point, then you must have $S(T) = 0$. In that case, we simply have $N(g_n) = n + 1$.

Turing's Theorem, the properties of Gram Points (and other factors) combine to allow Turing's Method, which we describe next. We begin with some key definitions.

- A Gram Point g_n is"good" if $(-1)^n Z(g_n) > 0$, and is "bad" otherwise.
- We define $[g_j, g_{j+1})$ as a *Gram interval.*
- For $k \ge 1$, we define $B_{j,k} = [g_j, g_{j+k})$ as a *Gram segment*; that is, the union of one or more consecutive Gram intervals.
- A *Gram block* is a Gram segment $B_{j,k} = [g_j, g_{j+k})$ such that g_j and g_{j+k} are good Gram Points and, if $k > 1$, $g_{j+1}, \cdots, g_{j+k-1}$ are bad Gram Points.
- We say a Gram segment $B_{j,k}$ satisfies *Rosser's Rule* if $Z(t)$ has exactly k zeros in $B_{j,k}$.

With those definitions, Turing's Method is based on the following theorem.

Assume the 3-tuple (A, B, t_0) satisfies Turing's Theorem. For $g_n > t_0$, let $B_{n,p} = [g_n, g_p)$ be a Gram segment that satisfies Rosser's Rule. Assume $B_{n,p}$ consists of K consecutive Gram blocks, where

$$K \ge \frac{B}{6\pi} (\log g_p)^2 + \frac{A - B \log 2\pi}{6\pi} \log g_p.$$

Then

$$N(g_n) \le n + 1 \quad \text{and} \quad N(g_p) \ge p + 1.$$

It is the $N(g_n) \le n + 1$ result that, in many circumstances, allows for an exact calculation of $N(T)$.

1.7 The Compiler Environment

We are using the gcc C compiler, which is part of the GNU Compiler Collection (GCC) – compilers for several programming languages, such as C, C++, Ada, and Fortran. These cross-platform compilers are available on Linux, macOS, and Windows, and many other operating systems.

In addition to the standard C libraries, we use three additional libraries, all common components in the toolchain of gcc-related development tools:

- **libgmp** – This GMP library is the GNU Multiple Precision Arithmetic Library, providing arbitrary-precision arithmetic.
- **libmpfr** – This MPFR library is the GNU MPFR Library. It is a C library for multiple-precision floating-point computations with correct rounding. MPFR is based on (and links to) the above GMP library. Our code uses (directly) only the MPFR library and not the GMP library.
- **libquadmath** – This is the GCC Quad-Precision Math Library. In a few circumstances where the MPFR library is not needed, we use this 128-bit floating point library.

The C code provided here should compile successfully on any of the GCC-supported operating systems, so long as the operating system is 64-bits.

We are using the gcc C compiler (and above libraries) on Windows 11. 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 required libraries 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.

1.8 Performance of the `HardyZWithCount` Library Function

We timed the performance of the `HardyZWithCount` library function. On our Intel Core i7-14700 Windows 11 system, we obtained the timings shown in the table below. The first column shows the initial t value. The second column shown the value of N in the formula for the main term of the Riemann-Siegel formula (see page 10) The third column shows the time (in seconds) to compute the Hardy Z value of the number in column one. The fourth column shows the time (in seconds) to compute eight Hardy Z values (t, $t+1$, ... $t+7$). The fifth column shows the time (in seconds) to compute eight Hardy Z values (t, $t+1$, ... $t+7$), using eight threads.

t	N	1	8	8 (threads)
10^6	398		0.024	0.007
10^7	1,261		0.067	0.014
10^8	3,989		0.209	0.034
10^9	12,615		0.642	0.089
10^{10}	39,894		2.008	0.264
10^{11}	126,156		6.290	0.865
10^{12}	398,942		19.850	2.565
10^{13}	1,261,566		62.577	8.554
10^{14}	3,989,422	24.758		30.832
10^{15}	12,615,662	78.175		114.577

Table 1.1: Timing the `HardyZWithCount` Library Function

Of course, the exact timing will vary run to run, but these times are good indicators of the approximate performance of the `HardyZWithCount` library function.

More than 99% of the running time is in computing the main term of the Riemann-Siegel formula. Because N (column two) increases by about $\sqrt{10} \approx 3.16$ for each 10 times increase in t, you should expect the calculating time to increase by about 3.16 times for each 10 times increase in t.