Numerical Precision and the Airy Functions

Elijan Jakob Mastnak
Student ID: 28181157

October 2020

Contents

Assignment

Create an efficient procedure for calculating the value of the Airy functions Ai and Bi on the entire real line with an absolute error less than 10^{-10} using a combination of the functions' Maclaurin series and asymptotic expansion. Repeat the procedure for relative error, and determine if an error less than 10^{-10} is feasible. Determine error using software capable of arbitrary-precision arithmetic, e.g. `Mathematica` or the Python packages `mpmath` or `decimal`.

Optional: Find the first 100 zeros $\{a_s\}_{s=1}^{100}$ and $\{b_s\}_{s=1}^{100}$ of the Airy functions Ai and Bi, respectively. Compare your results with the formulae in the theory section and to the results obtained with arbitrary-precision software.

1 Theory

This section might be superfluous—to jump directly to the solution, see Section 2.

The Airy functions Ai and Bi are the linearly independent solutions of the differential equation

$$y''(x) = -xy(x) = 0$$

The functions are expressed in the integral form as

$$\begin{aligned}\mathrm{Ai}(x) &= \frac{1}{\pi}\int_0^\infty \cos\left(\frac{t^3}{3} + xt\right) \mathrm{d}t \\ \mathrm{Bi}(x) &= \int_0^\infty \left[\exp\left(\frac{-t^3}{3} + xt\right) + \sin\left(\frac{t^3}{3} + xt\right)\right] \mathrm{d}t\end{aligned}$$

1.1 Maclaurin Series

For small $|x|$, the Airy functions are given by the Maclaurin series

$$\mathrm{Ai}(x) = \alpha f(x) - \beta g(x) \qquad \text{and} \qquad \mathrm{Bi}(x) = \sqrt{3}\left[\alpha f(x) + \beta g(x)\right] \tag{1}$$

where the coefficient α and β are

$$\begin{aligned}\alpha &= \mathrm{Ai}(0) = \frac{\mathrm{Bi}(0)}{\sqrt{3}} = \frac{1}{3^{2/3}\Gamma(\frac{2}{3})} \approx 0.355028053887817239 \\ \beta &= -\mathrm{Ai}'(0) = \frac{\mathrm{Bi}'(0)}{\sqrt{3}} = -\frac{1}{3^{1/3}\Gamma(\frac{1}{3})} \approx 0.258819403792806798\end{aligned}$$

and the functions f and g are

$$f(x) = \sum_{n=0}^{\infty} \left(\frac{1}{3}\right)_n \frac{3^n x^{3n}}{(3n)!} \qquad \text{and} \qquad g(x) = \sum_{n=0}^{\infty} \left(\frac{2}{3}\right)_n \frac{3^n x^{3n+1}}{(3n+1)!} \tag{2}$$

where $(z)_n$ is defined in terms of the gamma function as $(z)_n = \frac{\Gamma(z+n)}{\Gamma(z)}$.

1.2 Asymptotic Expansion

To approximate the Airy functions for large $|x|$, we introduce the new variable $\xi = \frac{2}{3}|x|^{3/2}$ and the asymptotic series L, P and Q

$$L(z) \sim \sum_{n=0}^{\infty} \frac{u_n}{z^n}, \qquad P(z) \sim \sum_{n=0}^{\infty} (-1)^n \frac{u_{2n}}{z^{2n}}, \qquad Q(z) \sim \sum_{n=0}^{\infty} (-1)^n \frac{u_{2n+1}}{z^{2n+1}} \tag{3}$$

where the coefficients u_n are defined in terms of the gamma function as

$$u_n = \frac{\Gamma\left(3n + \frac{1}{2}\right)}{54^n n! \Gamma\left(n + \frac{1}{2}\right)}$$

For large positive x, the Airy functions are approximated using L:

$$\mathrm{Ai}(x) \sim \frac{e^{-\xi}}{2\sqrt{\pi} x^{1/4}} L(-\xi) \qquad \text{and} \qquad \mathrm{Bi}(x) \sim \frac{e^{\xi}}{\sqrt{\pi} x^{1/4}} L(\xi) \tag{4}$$

while for large negative x, the Airy functions are approximated with P and Q:

$$\mathrm{Ai}(x) \sim \frac{1}{\sqrt{\pi}(-x)^{1/4}} \left[\sin\left(\xi - \frac{\pi}{4}\right) Q(\xi) + \cos\left(\xi - \frac{\pi}{4}\right) P(\xi)\right] \tag{5}$$

$$\mathrm{Bi}(x) \sim \frac{1}{\sqrt{\pi}(-x)^{1/4}} \left[\cos\left(\xi - \frac{\pi}{4}\right) Q(\xi) - \sin\left(\xi - \frac{\pi}{4}\right) P(\xi)\right] \tag{6}$$

1.3 Zeros

The first 100 zeros $\{a_n\}_{n=1}^{100}$ and $\{b_n\}_{n=1}^{100}$ of the Airy functions Ai and Bi, respectively, are well-approximated by the formulae

$$a_n = -f\left(\frac{3\pi(4n-1)}{8}\right) \qquad \text{and} \qquad b_n = -f\left(\frac{3\pi(4n-3)}{8}\right) \tag{7}$$

where the function f has the asymptotic expansion

$$f(z) \sim z^{2/3}\left(1 + \frac{5}{48} z^{-2} - \frac{5}{36} z^{-4} + \frac{77125}{82944} z^{-6} - \frac{108056875}{6967296} z^{-8} + \cdots\right)$$

1.4 A Motivating Example: Failure of Floating-Point Arithmetic

Here is a curiosity I though was worth mentioning: the constants α and β, used in the Maclaurin series for Ai and Bi, are given to 18 decimal places as

$$\alpha \approx 0.355028053887817239 \qquad \text{and} \qquad \beta \approx 0.258819403792806798$$

while the analytic formulae are

$$\alpha = \frac{1}{3^{2/3}\Gamma(\frac{2}{3})} \qquad \text{and} \qquad = -\frac{1}{3^{1/3}\Gamma(\frac{1}{3})}$$

An implementation of the analytic formulae using Python's built-in `math.gamma` function, which is supposed to return the "true" value of the gamma function but is limited by floating-point arithmetic, fails after 16 decimal places, while the arbitrary precision `mpmath.gamma` calculates α and β correctly, shown in the Python session below:

```python
import math
import mpmath
true_A = 0.355028053887817239     # true value of A = ai(0) to 18 decimal places
true_B = 0.258819403792806798     # true value of B = -ai'(0) to 18 decimal places

print("true A: {:.18f}".format(true_A))
print("mpmath A: " + mpmath.nstr(1/(mpmath.power(3, mpmath.fdiv(2,3)) *
↪  mpmath.gamma(mpmath.fdiv(2, 3))), mpmath.mp.dps))
print("system A: {:.18f}".format(1/(math.pow(3, 2/3) * math.gamma(2/3))))
print("true B: {:.18f}".format(true_B))
print("mpmath B: " + mpmath.nstr(-1/(mpmath.power(3, mpmath.fdiv(1,3)) *
↪  mpmath.gamma(mpmath.fdiv(1,3))), mpmath.mp.dps))
print("system B: {:.18f}".format(-1/(math.pow(3, 1/3) * math.gamma(1/3))))

>> true A:    0.355028053887817239    # true value
>> mpmath A:  0.355028053887817239    # mpmath gives correct result
>> system A:  0.355028053887817219    # system math fails at 16th decimal place!

>> true B:   -0.258819403792806798
>> mpmath B: -0.258819403792806798
>> system B: -0.258819403792806768    # system math fails at 16th decimal place!
```

Clearly, floating-point arithmetic has its limitations!

2 Solution

2.1 Modifications to the Auxiliary Functions f and g

Calculating the series for f and g directly with Equations 2 is inefficient, because one must continually calculate the factorials and powers "from scratch". The problem is solved by writing f and g in recursive form and calculating successive terms recursively[1].
To do this, we first simplify $(z)_n$ using the gamma function identity

$$(z)_n \equiv \frac{\Gamma(z+n)}{\Gamma(z)} = z(z+1)\cdots(z+n-1)$$

for all positive integers $n \in \mathbb{N}^+$. The first few terms of f and g for $n = 0, 1, 2, 3$ are then

$$\begin{array}{llllll} f\colon & 1 & \frac{1}{3}\cdot\frac{3x^3}{3!} & \frac{1}{3}\left(\frac{1}{3}+1\right)\frac{3^2x^6}{6!} & \frac{1}{3}\left(\frac{1}{3}+1\right)\left(\frac{1}{3}+2\right)\frac{3^3x^9}{9!} \\ g\colon & x & \frac{2}{3}\cdot\frac{3x^4}{4!} & \frac{2}{3}\left(\frac{2}{3}+1\right)\frac{3^2x^7}{7!} & \frac{2}{3}\left(\frac{2}{3}+1\right)\left(\frac{2}{3}+2\right)\frac{3^3x^{10}}{10!} \end{array}$$

A close look at the sequences reveals the recursive patterns (for $n = 1, 2, \ldots$)

$$f_{n+1}(x) = \left(\frac{1}{3}+n-1\right)\frac{3x^3}{3n(3n-1)(3n-2)}f_n(x), \qquad f_0 = 1$$

$$g_{n+1}(x) = \left(\frac{2}{3}+n-1\right)\frac{3x^3}{(3n+1)(3n)(3n-1)}g_n(x), \qquad g_0 = x$$

A basic implementation in Python—with truncation logic omitted—might read

[1]Another benefit: fewer calculations because of the recursive implementation–besides being faster–should also reduce the accumulation of floating-point arithmetic errors.

```python
def f_and_g_example(x):
    next_term_f = 1     # next term in series, helps store recursive calculations
    next_term_g = x     # note g starts with initial value x and f with 1
    sum_f = next_term_f  # cumulative sum
    sum_g = next_term_g

    for n in range(1, max_iterations):   # truncation logic implemented separately
        next_term_f *= (1/3 + n - 1) * 3 * (x**3) / ((3*n) * (3*n - 1) * (3*n - 2))
        next_term_g *= (2/3 + n - 1) * 3 * (x**3) / ((3*n + 1) * (3*n) * (3*n - 1))
        current_sum_f += next_term_f     # update cumulative sum
        current_sum_g += next_term_g     # update cumulative sum
```

With f and g implemented, the Airy functions can then be found in the Maclaurin series regime with a straightforward implementation of Equation 1. For example:

```python
def airy_maclaurin(x):
    f = f(x)
    g = g(x)
    ai = A * f - B * g
    bi = math.sqrt(3) * (A*f + B*g)
```

2.2 Modifications to Asymptotic Series

Again, we aim to write the series L, P and Q recursively to improve efficiency and minimize potential for floating-point arithmetic errors. First, we use the gamma function identities

$$\Gamma\left(n+\tfrac{1}{2}\right) = \frac{\sqrt{\pi}(2n)!}{4^n n!} = \frac{\sqrt{\pi}(2n-1)!!}{2^n} \implies \Gamma\left(3n+\tfrac{1}{2}\right) = \frac{\sqrt{\pi}(6n-1)!!}{2^{3n}}$$

to rewrite the u_n term as

$$u_n = \frac{\Gamma\left(3n+\frac{1}{2}\right)}{54^n n! \Gamma\left(n+\frac{1}{2}\right)} = \frac{(6n-1)!!}{4^n 54^n n! (2n-1)!!}, \qquad n = 1, 2, \ldots$$

The first few u_n terms for $n = 0, 1, 2, 3$ are

$$u_0 = 1 \qquad u_1 = \frac{5\cdot 3}{4\cdot 54} \qquad u_2 = \frac{11\cdot 9\cdot 7\cdot 5\cdot 3}{(4\cdot 54)^2(2)(3)} \qquad u_3 = \frac{17\cdot 15\cdot 13\cdot 11\cdot 9\cdot 7\cdot 5\cdot 3}{(4\cdot 54)^3(3\cdot 2)(5\cdot 3)}$$

Examining the sequence reveals the recursive pattern

$$u_{n+1} = \frac{(6n-1)(6n-3)(6n-5)}{(4\cdot 54)\cdot n\cdot(2n-1)} u_n, \qquad u_0 = 1$$

This recursive formula for u_{n+1} makes it possible to rewrite L, P and Q recursively:

- If $L_n(x)$ denotes the Lth term in the series, then for $n = 1, 2, 3, \ldots$

$$L_{n+1} = \frac{(6n-1)(6n-3)(6n-5)}{216n(2n-1)x} L_n, \qquad L_0 = 1$$

- For P, for $n = 1, 2, 3, \ldots$ the recursive formula is

$$P_{n+1}(x) = -\frac{(12n-1)(12n-3)\cdots(12n-11)}{216^2(2n)(2n-1)(4n-1)(4n-3)x^2} P_n(x), \qquad P_0 = 1$$

- And for Q, for $n = 1, 2, 3, \dots$ the recursive formula is

$$Q_{n+1}(x) = -\frac{(12n+5)(12n+3)\cdots(12n-5)}{216^2(2n+1)(2n)(4n+1)(4n-1)x^2}Q_n, \qquad Q_0 = \frac{u_1}{z} = \frac{15}{216x}$$

In Python, a basic implementation of L, P and Q might read

```python
def L_P_Q_example(x):
    next_term_L = 1  # next term in the series, helps store recursive calculations
    next_term_P = 1             # initial value is 1 for L and P...
    next_term_Q = 15/(216 * x)  #... and 15/(216 * x) for Q
    sum_L = next_term_L         # cumulative sum
    sum_P = next_term_P
    sum_Q = next_term_Q

    for n in range(1, max_iterations):    # truncation logic implemented separately
        next_term_L *= (6*n - 1)*(6*n - 3)*(6*n - 5)/(216 * n * (2*n-1) * x)
        next_term_P *= - (12*n - 1)*(12*n - 3)*(12*n - 5)*(12*n - 7)*(12*n -
        ↪  9)*(12*n - 11)/((216 ** 2) * (2*n) * (2*n - 1) * (4*n - 1) * (4*n - 3)
        ↪  * (x ** 2))
        next_term_Q *= -(12*n + 5)*(12*n + 3)*(12*n + 1)*(12*n - 1)*(12*n -
        ↪  3)*(12*n - 5)/((216 ** 2) * (2*n) * (2*n + 1) * (4*n + 1) * (4*n - 1) *
        ↪  (x ** 2))

        sum_L += next_term_L    # update cumulative sums...
        sum_P += next_term_P
        sum_Q += next_term_Q
```

With L, P and Q implemented, the Airy functions can then be found in the asymptotic regime using Equations 4, 5 and 6 and the new variable $\xi = \frac{2}{3}|x|^{3/2}$, as shown in Subsection 1.2. For example:

```python
def airy_asym_negative(x):
    xi = 2 / 3 * math.pow(abs(x), 1.5)
    q = Q(xi, x)
    p = P(xi, x)
    ai = (math.sin(xi - (math.pi/4))*q + math.cos(xi - (math.pi/4))*p) /
    ↪  (math.sqrt(math.pi) * math.pow(-x, 0.25))
    bi = (math.cos(xi - (math.pi/4))*q - math.sin(xi - (math.pi/4))*p) /
    ↪  (math.sqrt(math.pi) * math.pow(-x, 0.25))

def airy_asym_positive(x):
    xi = 2/3 * math.pow(abs(x), 1.5)
    ai = L(-xi, x) * math.exp(-xi) / (2 * math.sqrt(math.pi) * math.pow(x, 0.25))
    bi = L(xi, x) * math.exp(xi) / (math.sqrt(math.pi) * math.pow(x, 0.25))
```

2.3 Truncation

2.3.1 Truncating the Maclaurin Series

I truncated the series for $f(x)$ and $g(x)$ either when subsequent terms differed by less than 10^{-10} or after 40 terms, whichever came first. In practice, I reached the 10^{-10} limit at around 20 to 30 terms near the Maclaurin regime endpoints and in as few as 3 to 5 terms near zero, and I never actually reached 40 terms. A Python implementation follows:

```python
'''Example of truncation implementation for the series f(x)'''
def f(x):
  epsilon = 1e-10
  next_term = 1
  current_sum = next_term
  for n in range(1, 40):  # at most 40 terms
      old_term = next_term
      next_term *= (1/3 + n - 1) * 3 * (x**3) / ((3*n) * (3*n - 1) * (3*n - 2))
      current_sum += next_term    # update cumulative sum
      if abs(old_term - next_term) < epsilon: break  # exit when subsequent terms
      ↪  become arbitrarily close
  return current_sum
```

I also tested truncating when the partial sums differed by less than 10^{-10}, i.e. when `abs(old_sum - current_sum) < epsilon`, but this method resulting in larger erros than comparing subsequent terms.

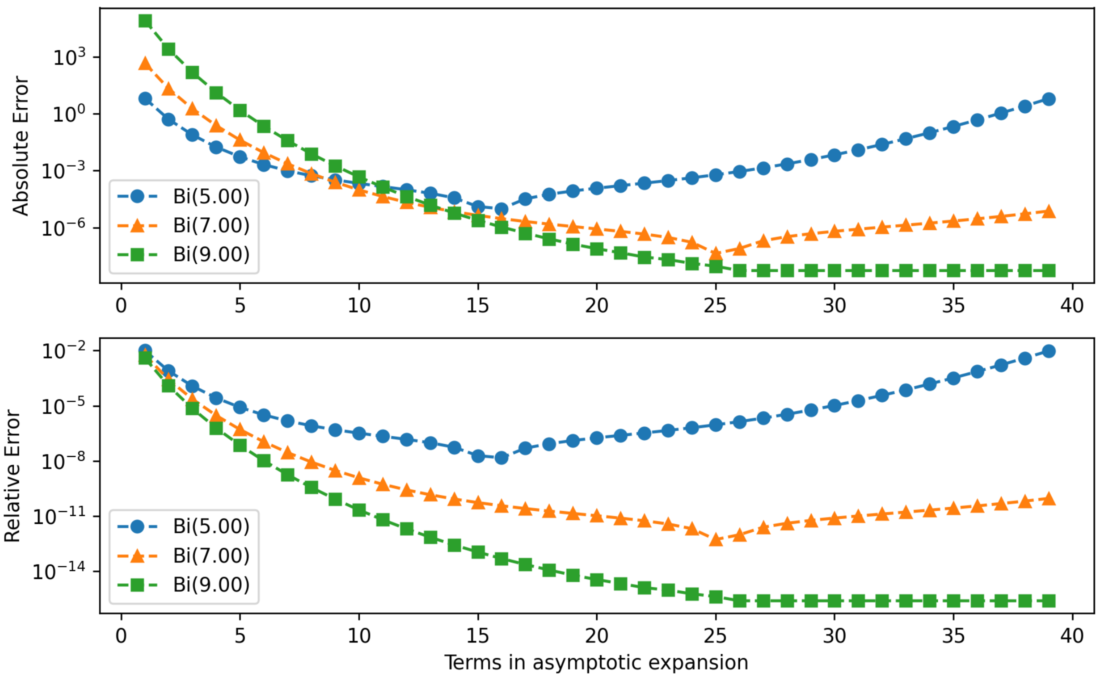

Figure 1: Absolute and relative error of Bi(x) for positive x as a function of the number of terms in the asymptotic expansion. Note that for small x, adding terms causing error to increase, as expected for an asymptotic series.

2.3.2 Truncating the Asymptotic Series

I truncated the series for $L(x), P(x)$ and $Q(x)$ (see Equations 3) using three conditions:

1. If subsequent terms differed by less than 10^{-16}. Why so much lower than 10^{-10}? I wanted to push the limits of the accuracy I could achieve, and 10^{-16} reduced both absolute and relative error in the large x regime by multiple orders of magnitude compared to 10^{-10}.

2. If subsequent terms in the expansion began to increase, indicating divergence. This applied largely to the smaller values of $|x|$ in the asymptotic regime; see Figure 1.

3. If the calculation reached 40 terms in the asymptotic expansion. This rarely happened; terms began to differ by less than 10^{-16} after 20-30 iterations for large $|x|$.

I feel like there's enough code already, so I'm leaving out this example; the syntax and logic is analogous to the Maclaurin truncation implementation given above.

2.4 Switching Between Asymptotic and Maclaurin Regimes

Obviously, the asymptotic series better approximate Ai and Bi for large $|x|$, while the Maclaurin series works better for small $|x|$. I chose the breakpoints between regimes so that error as a function of x changed continuously across the transition. I couldn't find a single satisfactory value for the Maclaurin-to-asymptotic transition for positive x, so I implemented separate breakpoints for Ai and Bi. The implementation I used is:

```python
asym_to_power = -7      # from negative asymptotic to Maclaurin
power_to_asym_A = 5.5   # from Maclaurin to positive asymptotic for Ai
power_to_asym_B = 8.2   # from Maclaurin to positive asymptotic for Bi

if x <= asym_to_power:                          # asymptotic for both A and B
    # calculate Ai and Bi...
elif asym_to_power < x <= power_to_asym_A:      # power for both A and B
  # calculate Ai and Bi...
elif power_to_asym_A < x <= power_to_asym_B:    # asymptotic for A and power for B
    # calculate Ai and Bi...
elif x > power_to_asym_B:                       # asymptotic for both A and B
    # calculate Ai and Bi...
```

2.5 Evaluating Error

Figure 2 shows the relative and absolute errors of my implementation of Ai and Bi.
I used `mpmath`'s arbitrary precision `airyai` and `airybi` functions as the "true" value of the Airy functions when evaluating the error of my own implementation. I set `mpmath` to use 18 decimal places, corresponding to a mantissa precision of 63 bits, compared to the 52-bit mantissa precision of IEEE 745 double-precision values. I could have used higher precision as a better reference, but that seems superfluous, since a 63-bit mantissa is already well-above the 52-bit precision floating-point arithmetic I am working with.

2.6 Results and Takeaway

Figure 3 shows the graphs of the Ai and Bi implementation for $x \in [-15, 1]$.
Reflecting on the results summarized in Figures 2 and 3, I'm satisfied with:

- Absolute and relative error in the asymptotic regime of large negative x, which are both comfortably below 10^{-10}.
- Absolute error of the Maclaurin series for small $|x|$, which remains comfortably below 10^{-10} in the Maclaurin regime.
- Aside from a slight divergence for Ai near the positive Maclaurin-to-asymptotic transition (discussed below), relative error in both Ai and Bi remains below 10^{-10} over the entire real line.

I'm not satisfied with:

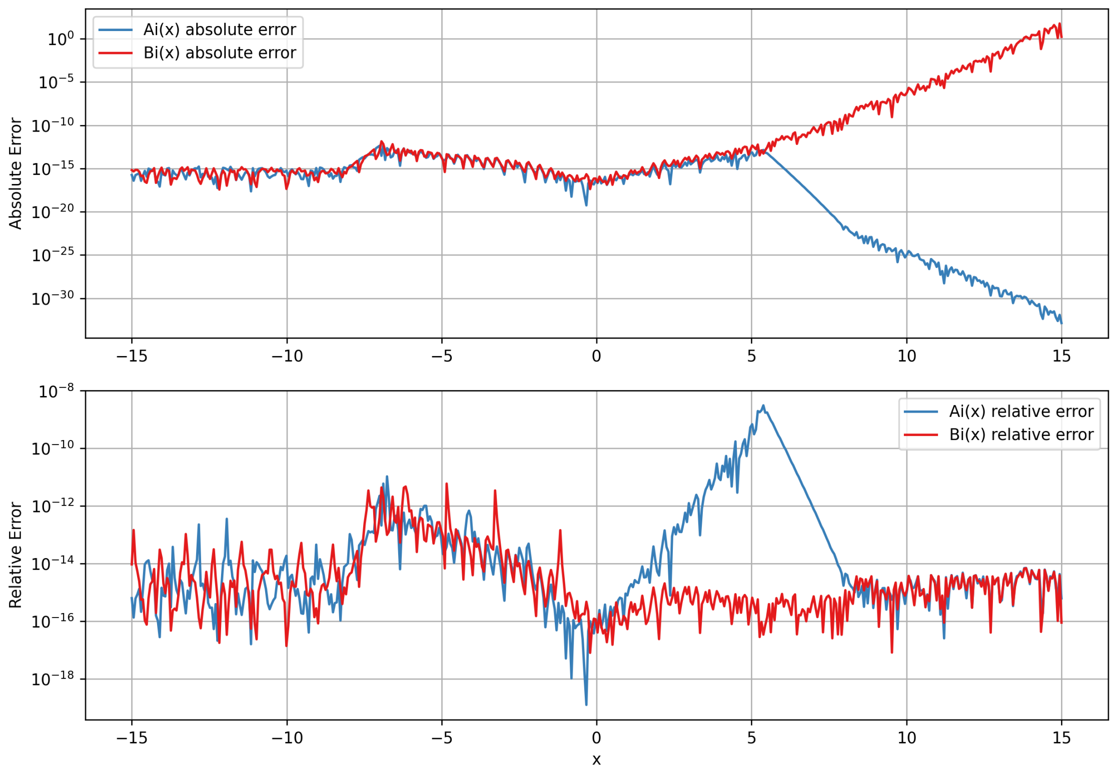

Figure 2: Relative and absolute error of my implementation of the functions Ai and Bi. Both relative and absolute error stay well below 10^{-10} for negative x, but absolute error for Bi diverges for large x. See Subsection 2.6 for a more thorough discussion.

- For $\mathrm{Ai}(x)$, relative error climbs to roughly $3 \cdot 10^{-9}$ in the neighborhood of the transition point $x = 5.5$ both in the Maclaurin and asymptotic approximations. I aimed to keep relative error below 10^{-10} over the entire real line, and was unsuccessful because of this trouble spot.
- Divergence of the absolute error of $\mathrm{Bi}(x)$ for large positive x, which means I was unsuccessful in the the assignment, which called for absolute error below 10^{-10} over the entire real line. This error divergence appears to be a consequence of finite-precision floating-point arithmetic, which becomes problematic as Bi diverges towards $+\infty$. It is worth noting, however, that relative error of Bi remains well below 10^{-10} over the entire real line.

3 Zeros

I calculated the Airy functions' zeros using three methods:

1. The arbitrary precision `mpmath` functions `airyaizero(n)` and `airybizero(n)`, which return Ai and Bi's nth zero, respectively, to arbitrary precision.
2. The asymptotic formula given in Equation 7.
3. My own algorithm using the bisection method and my implementation of Ai and Bi.

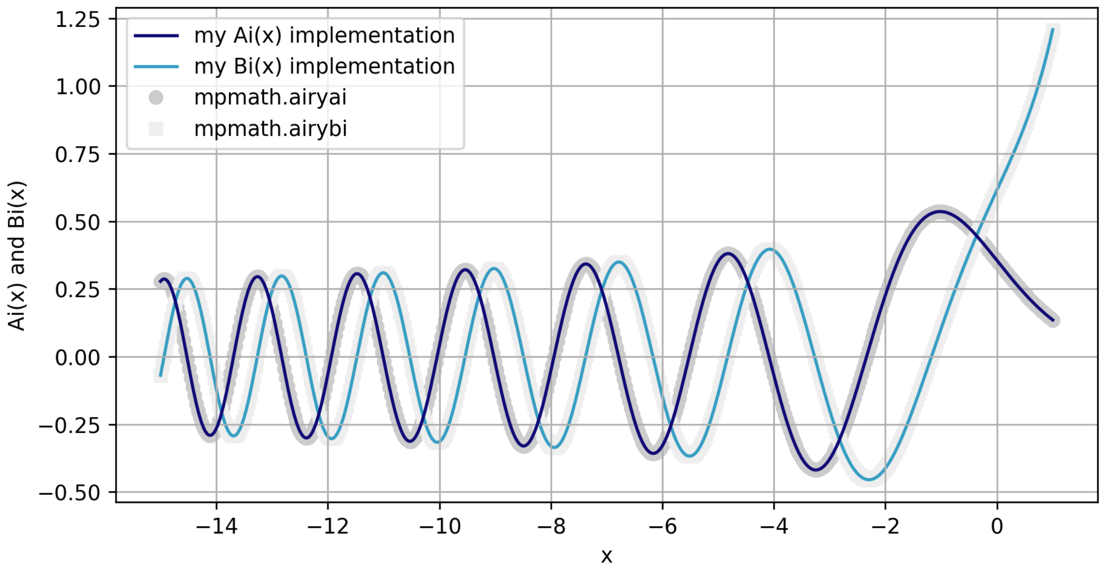

Figure 3: Graphs of my implementation of Ai and Bi with arbitrary precision values from `mpmath.airyai` and `mpmath.airybi` for reference.

3.1 Asymptotic Zero Algorithm

I directly implemented Equation 7. The implementation is straightforward and I think including code would be overkill. The only detail worth mentioning is that for the first zero, I truncated the asymptotic series at the fourth term for Ai and the second term for Bi to minimize error. This improved the accuracy by roughly one and two orders of magnitude, respectively.

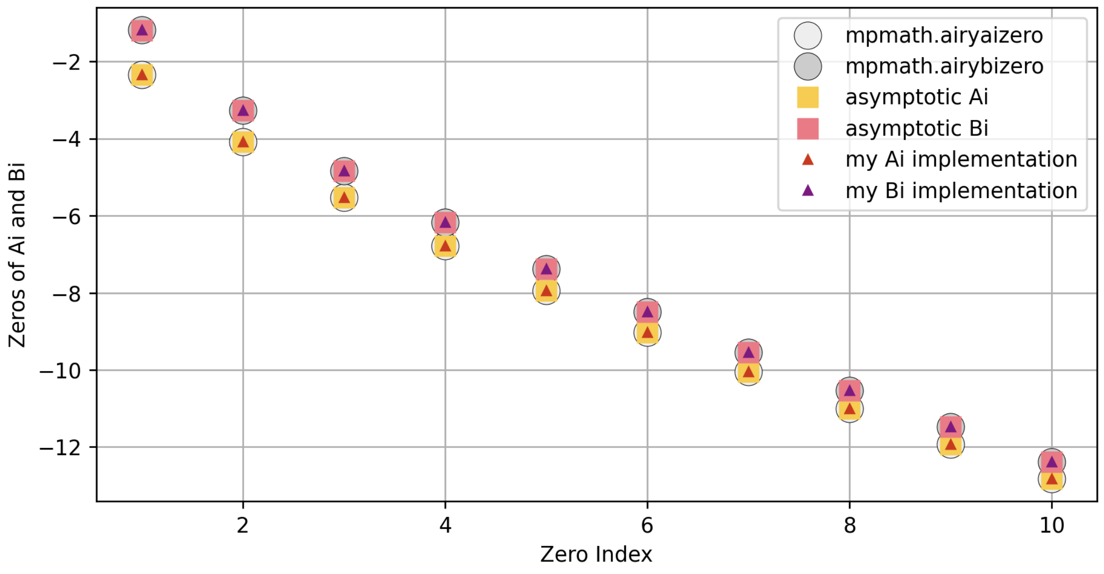

Figure 4: The first 10 zeros of Ai and Bi found using `mpmath.airyzero`, the asymptotic formula from Equation 7 and my bisection method implementation from Subsection 3.2. Only 10 zeros are shown to reduce clutter; for the full 100 zeros see Figure 5.

3.2 My Attempt at a Zero-Finding Algorithm

The algorithm actually works satisfactorily. It can be abstracted into three steps:

1. Calculate Airy function values on the interval containing the desired number of zeros, e.g. $[-61, 0]$ for the first 100 zeros, using my Airy function implementation.

2. Loop through the Airy function values and detect when the values change sign, meaning there is a zero in the interval between the two function values. Store the x values of the interval endpoints in an array.

   The sample rate of the function values must be large enough to have at least one point between each zero—500 points is more than enough to cover the first 100 zeros.

3. Loop through the array of intervals containing zeros and find each zero using the bisection method: pass the left and right endpoints as arguments, specify a tolerance, and calculate function values using my Airy function implementation.

In Python, a possible implementation of the bisection zero algorithm reads

```python
max_zeros = 100                     # find 100 zeros
eps = 1e-8                          # tolerance for bisection method
X = np.linspace(-65, 0, 500)        # sample 500 points between -65 and 0
my_ai_values = get_my_ai_values(X)  # calculate values of Ai on X
current_value = my_ai_values[0]     # value of Ai at first point
zero_intervals = []   # intervals representing the two closest points to each zero
zero_counter = 0      # counts how many zeros have been found
for i in range(1, numpoints):  # loop through Airy function values
    prev_value = current_value         # store previous value
    current_value = my_ai_values[i]    # update current value

    if zero_counter < max_zeros and ((prev_value <= 0 <= current_value) or
    ↪  (current_value <= 0 <= prev_value)):  # crossed a zero
        zero_intervals.append((X[i-1], X[i]))  # record interval containing zero
        zero_counter += 1                      # update zero counter

my_airyaizeros = []                         # array to store zeros
for i in range(0, len(zero_intervals)):    # loop through array of zero intervals
    zero = bisection(my_airyai, zero_intervals[i][0], zero_intervals[i][1], eps)
    my_airyaizeros.append(zero)    # calculate zero with bisection, store in array
```

The bisection implementation I used is:

```python
def bisection(f, a, b, eps):
    if a == 0.0: return a   # safety check in case endpoints are zeros
    if b == 0.0: return b

    counter = 0
    max_iterations = 100  # infinite loop protection; never reached in practice
    while abs(b-a) > eps:
        c = a + (b-a)/2   # less risk of overflow than (a+b)/2

        if np.sign(f(a)) == np.sign(f(c)): a = c   # shift left endpoint to c
        else: b = c                                # shift right endpoint to c
        counter += 1
        if counter > max_iterations: break  # infinite loop protection

    return a + (b - a)/2
```

The algorithm's accuracy depends largely on the tolerance ϵ used with the bisection method, as seen in Figure 5. Of course, the accuracy can be no better than the accuracy of my Airy function implementation, which has an absolute error of about 10^{-15} for large negative x.
Figure 5 includes a run of the bisection method with an excessively small tolerance of $\epsilon = 10^{-20}$. This is done intentionally to demonstrate that floating-point precision limits the algorithm's accuracy to about 10^{-16}, regardless how small I make ϵ.

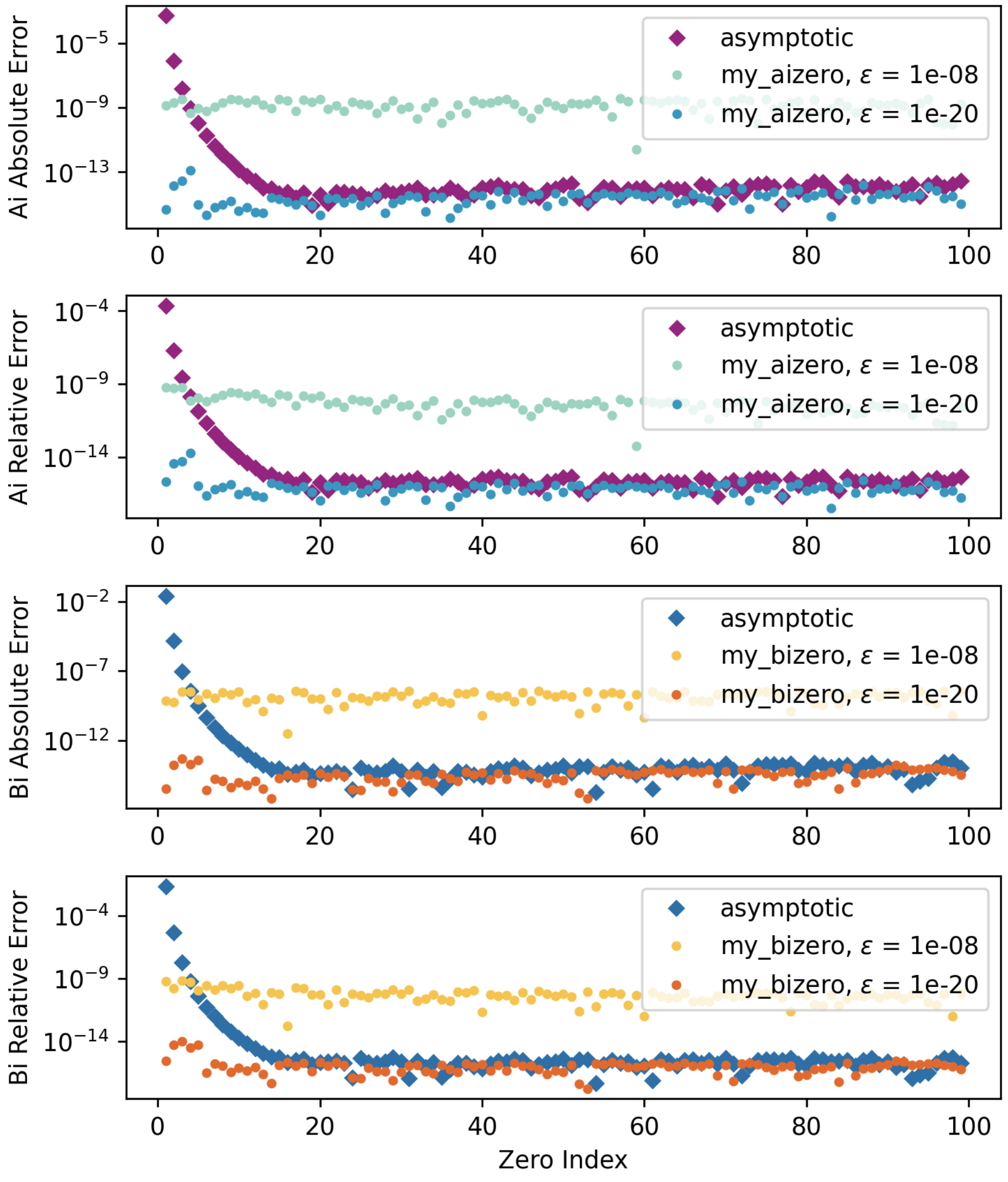

Figure 5: Absolute and relative errors of the first 100 zeros of Ai and Bi using the asymptotic formula from Equation 7 and two runs of my own bisection method implementation, discussed in Subsection 3.2. The accuracy of the asymptotic approximation exponentially improves with increasing zero index, while the bisection implementation accuracy is a largely constant function of bisection tolerance ϵ. *The second bisection run with excessively low $\epsilon = 10^{-20}$ shows that floating-point precision limits accuracy to about 10^{-15}.*