Lab2 Report

Zirui Wan (Question 1 and 3), Rundong Zhou (Question 2)

September 25, 2020

1 Question 1

1.1 Q1b

According to the output of the program, compare numerical derivative and absolute value of error side-by-side:

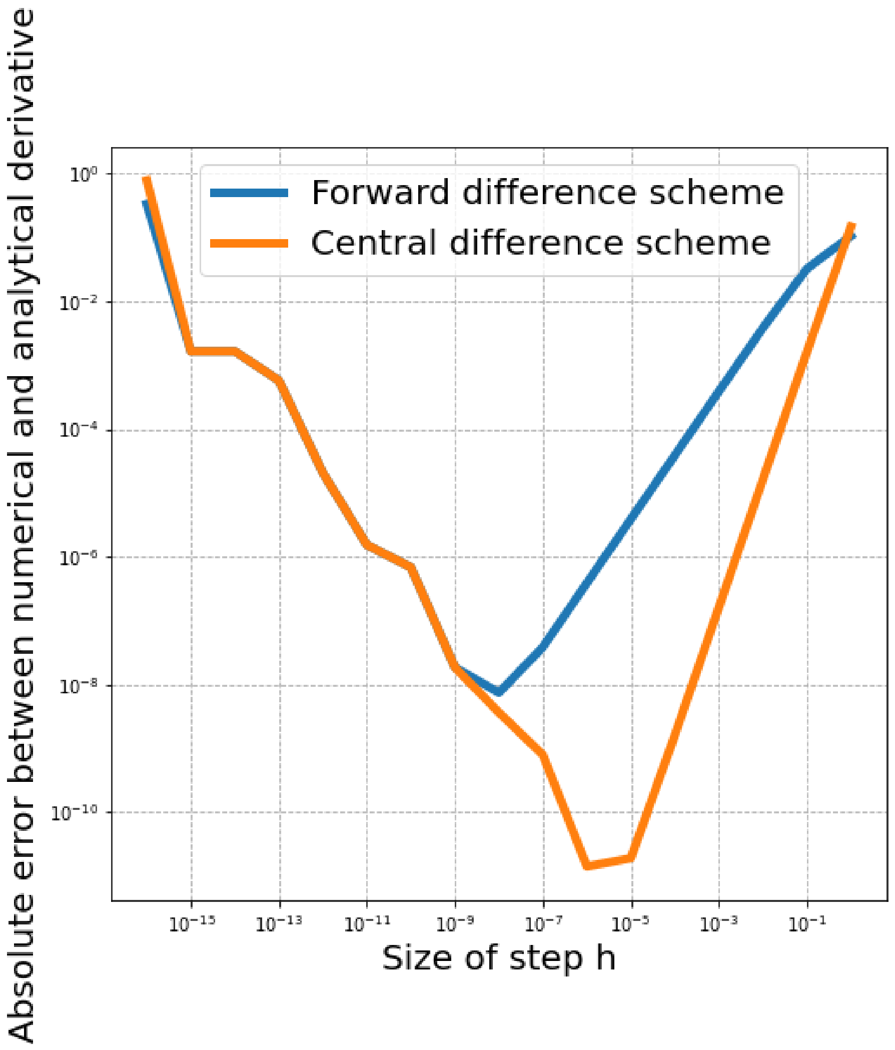
Step size	Numerical f'	Error
10^{-16}	-1.11022302	3.31×10^{-1}
10^{-15}	-0.77715612	1.64×10^{-3}
10^{-14}	-0.77715612	1.64×10^{-3}
10^{-13}	-0.77937656	5.76×10^{-4}
10^{-12}	-0.77882145	2.07×10^{-5}
10^{-11}	-0.77879925	1.54×10^{-6}
10^{-10}	-0.77880147	6.85×10^{-7}
10^{-9}	-0.7788008	1.86×10^{-8}
10^{-8}	-0.77880079	7.45×10^{-9}
10^{-7}	-0.77880082	3.85×10^{-8}
10^{-6}	-0.77880117	3.89×10^{-7}
10^{-5}	-0.77880468	3.89×10^{-6}
10^{-4}	-0.77883972	3.89×10^{-5}
10^{-3}	-0.77918953	3.89×10^{-4}
10^{-2}	-0.78262986	3.83×10^{-3}
10^{-1}	-0.81124457	3.24×10^{-2}
10^{0}	-0.67340156	1.05×10^{-1}

Before step size tends to 10^{-8} which yields the smallest error, absolute value of error decreases, after that error increases.

1.2 Q1c

1.2.1 Plot of Q1c and Q1d

1.2.2 Comments



The error from forward difference scheme is shown above. The error could be formulated as:

$$\epsilon = \frac{2C|f(x)|}{h} + \frac{1}{2}h|f''(x)| \tag{1}$$

This equation consists of two terms, where the first decrease with h and the second increases with h. Thus, there must exists some global minimum, which is 10^{-8}. Before this global minimum, the first term dominates, which is the rounding error because it's associated with the accuracy of the computer (constant C). After the global minimum the second term dominates, which is the truncation error caused by the approximation of the infinite process of derivative calculation.

1.3 Q1d

1.3.1 Comments

The error from central difference scheme is plotted on the same figure above. You can see that, when the rounding error dominates, it has the same accuracy as the forward method. However, it is able to archive higher accuracy when the truncation error starts to dominate the forward method. After the central method reaches its optimal accuracy, the truncation error will dominate it too and the performance starts to degrade faster than the forward method. Eventually, when the step size is relatively large, the forward method will outperform the central method.

2 Question 2

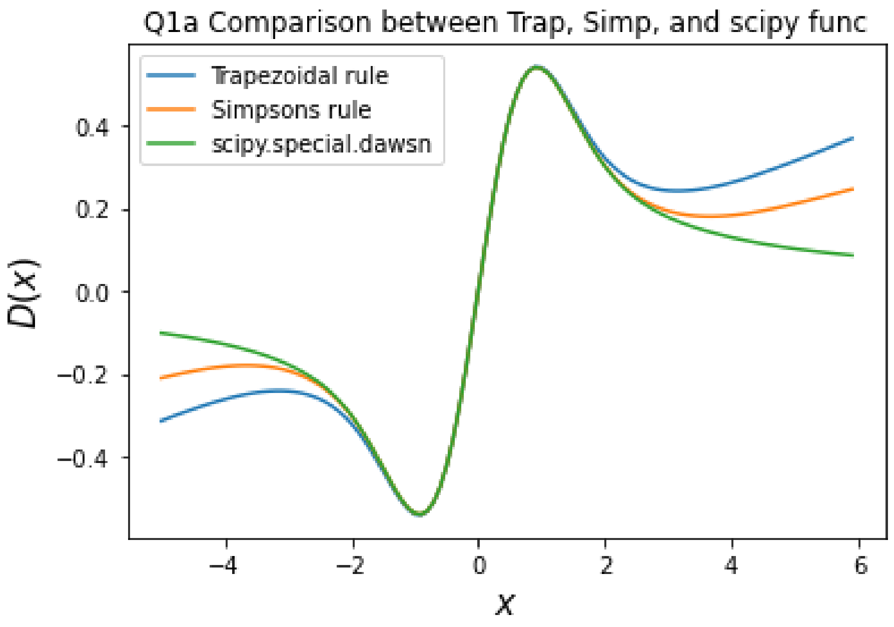
2.1 Q2a

2.1.1 i

According to the out put of my program:

```
D(4), Trapezoidal rule:  0.26224782053479523
D(4), Simpsons rule:  0.1826909645971217
D(4), Scipy special function:  0.1293480012360051
Difference between Trapesoidal rule & Scipy special function:  0.13289981929879013
Difference between Simpsons rule & Scipy special function:  0.0533429633611166
```

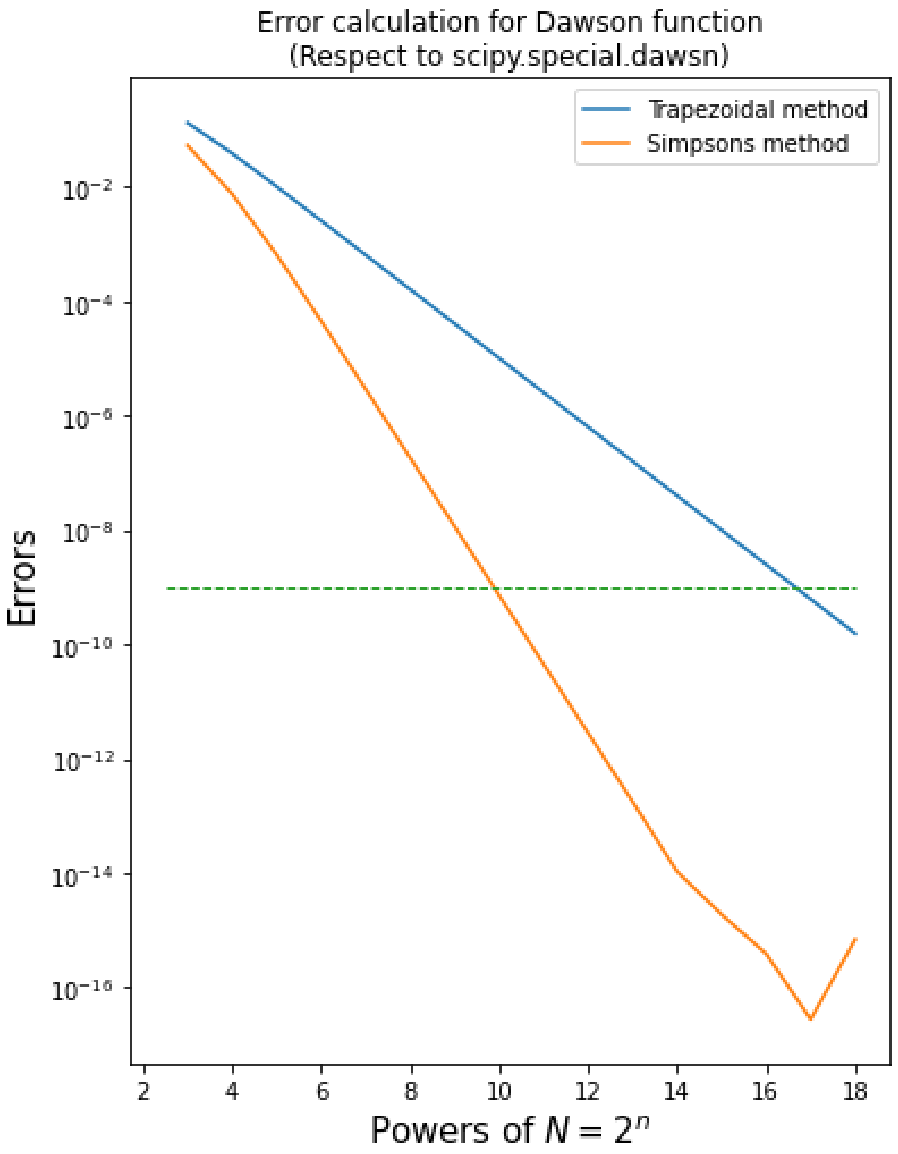
We can tell that both Trapezoidal and Simpson's rule show a significant difference from `scipy.special.dawsn` at $x = 4$. While the difference for Simpson's rule is about two times smaller than Trapezoidal rule. We can also see this from the plot:



Three methods tend to diverge at around $x = \pm 2$. The differences are quite significant at $D(4)$.

2.1.2 ii

So I plot the errors (with respect to `scipy.special.dawsn`) for both Trapezoidal and Simpson's methods from $n = 2$ up to $n = 18$.



As we can tell from the graph, Trapezoidal method hits $\mathcal{O}(10^{-9})$ mark (the horizontal green dot line on the graph) at around $n = 17, N = 2^{17}$. Error for Simpson's rule decreases more steeply and hit the mark at around $n = 10, N = 2^{10}$. Also we can notice that line for Simpson's rule turns messy when the error hits $\mathcal{O}(10^{-15})$. This corresponds to the computer accuracy limit mentioned during the lecture, which is also $\mathcal{O}(10^{-15})$.
For the time consumption, I use `for` loop to run both methods at desired accuracy ($n = 17$ for Trapezoidal and $n = 10$ for Simpson's) as well as `scipy.special.dawsn` for 50 times, and compute the average:

```
Average Time consumption, Trapezoidal method, n=17:  0.1406150197982788s
Average Time consumption, Simpsons method, n=10:  0.0012566328048706054s
Average Time consumption, scipy.special.dawns:  0.0s
```

From the result we can tell that, at the same accuracy, Simpson's method takes about 1 out of 100 less time than Trapezoidal method. The `scipy.special.dawsn` runs so fast that `time()` cannot tell the actual time consumption.

2.1.3 iii

According to the textbook method, I calculate the error for both Trapezoidal rule and Simpson's rule according to the formula, with $x = 4$ and $N_1 = 32, N_2 = 64$:

$$\epsilon_2 = \frac{1}{3}(I_2 - I_1) \text{ (Trapezoidal rule)}$$

$$\epsilon_2 = \frac{1}{15}(I_2 - I_1) \text{ (Simpson's rule)}$$

My program gives the following result:

```
Error of Trapezoidal method, N1=32 N2=64: 0.0025465686529556795
Error of Simpsons method, N1=32 N2=64: 4.115768458675488e-05
```

The result is pretty close to the previous error calculation chart done in part ii:

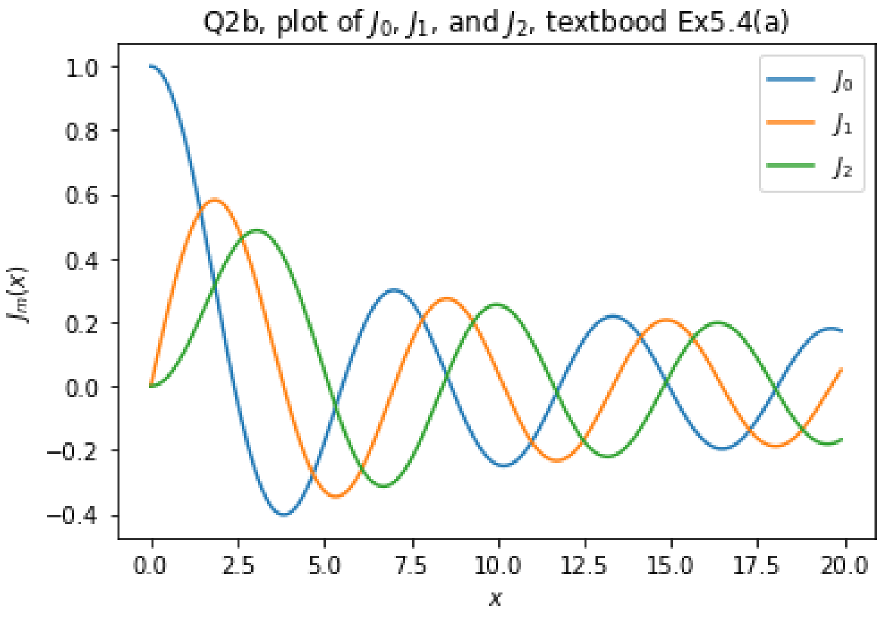
$$N = 64 = 2^5, n = 5 \leftrightarrow \mathcal{O}(10^{-2}) \text{ (Trapezoidal rule)}$$

$$N = 64 = 2^5, n = 5 \leftrightarrow \mathcal{O}(10^{-4}) \text{ (Simpson's rule)}$$

2.2 Q2b

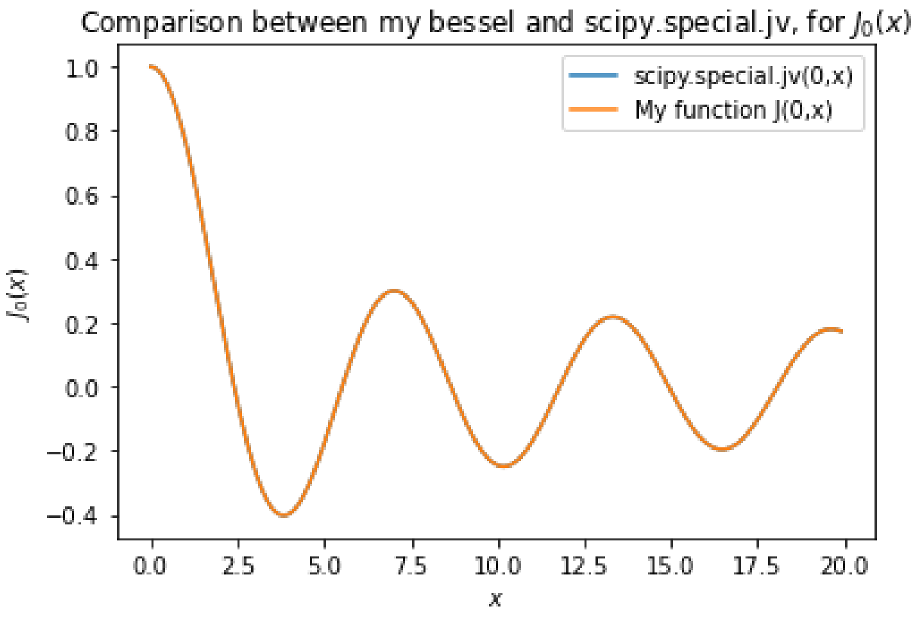
2.2.1 Exercise 5.4(a)

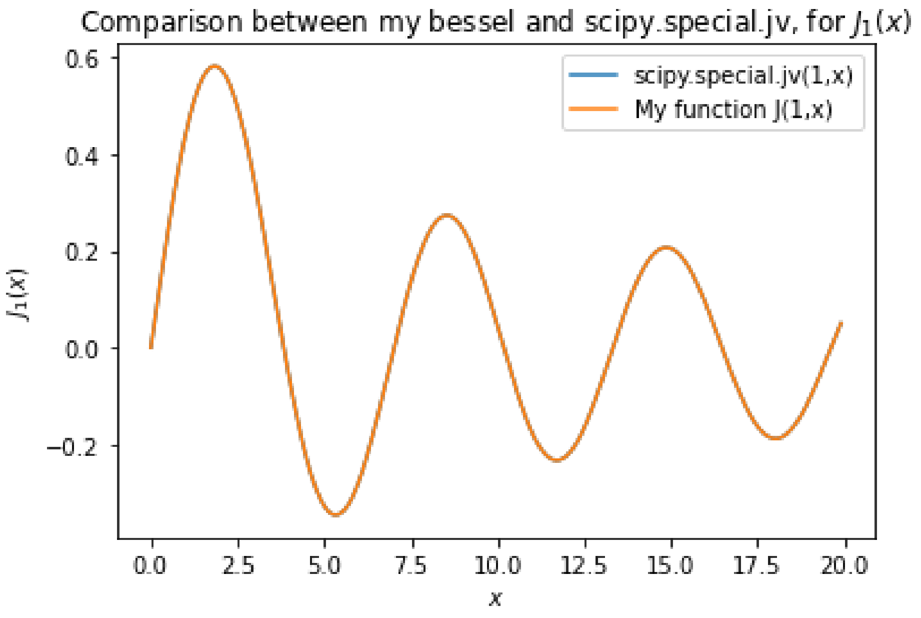
So the plot of the Bessel functions J_0, J_1, and J_2 as a function of x from $x = 0$ to $x = 20$ using Simpson's rule with $N = 1000$ is shown below:



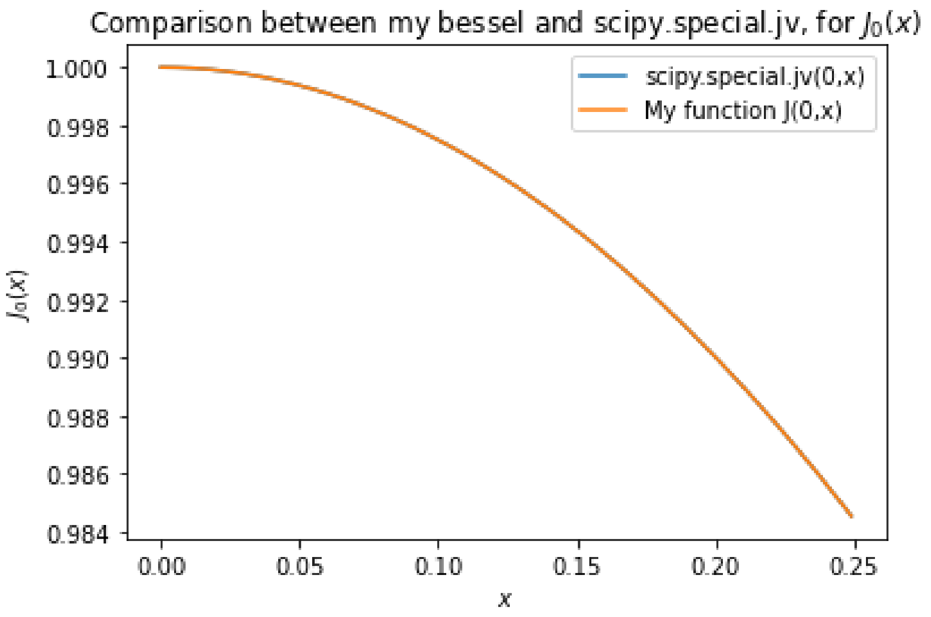
2.2.2 Comparison between my Bessel functions and `scipy.special.jv`

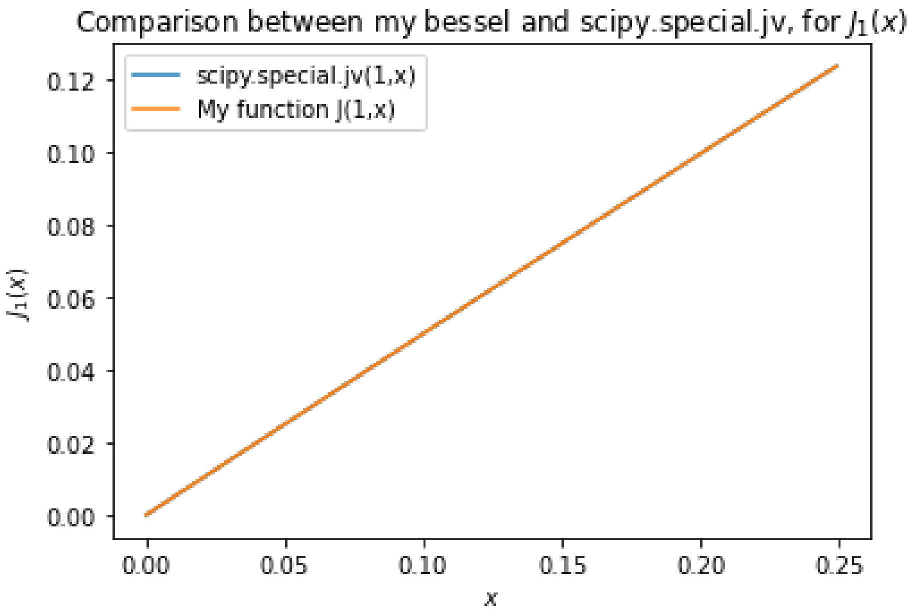
So I first plot J_0 and J_1 from $x = 0$ to $x = 20$ to compare my Bessel function with `scipy.special.jv`:





The graph looks surprisingly good. My Bessel function overlaps `scipy.special.jv` perfectly. It is 'too' good. Since I don't trust myself, I zoom in and plot J_0 and J_1 again from $x = 0$ to $x = 0.25$:





Perfect overlapping again!
I doubt myself even more, there must be something wrong in my code! But I cannot find anything. I add these lines into my code to see what is the actual difference between my Bessel function and `scipy.special.jv`:

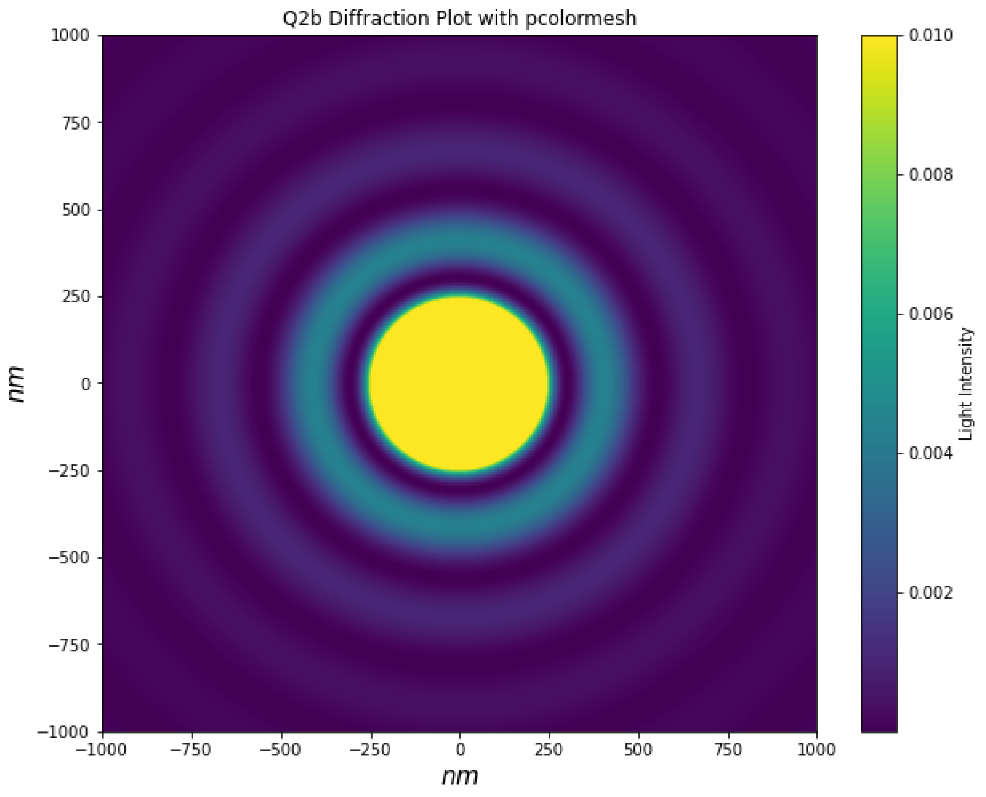
```
print('Difference between my Bessel and scipy.special.jv with m=0, x=0:',
      J(0,0) - special.jv(0,0))
print('Difference between my Bessel and scipy.special.jv with m=2, x=2:',
      J(2,2) - special.jv(2,2))
print('Difference between my Bessel and scipy.special.jv with m=1, x=2.5:',
      J(1,2.5) - special.jv(1,2.5))
```

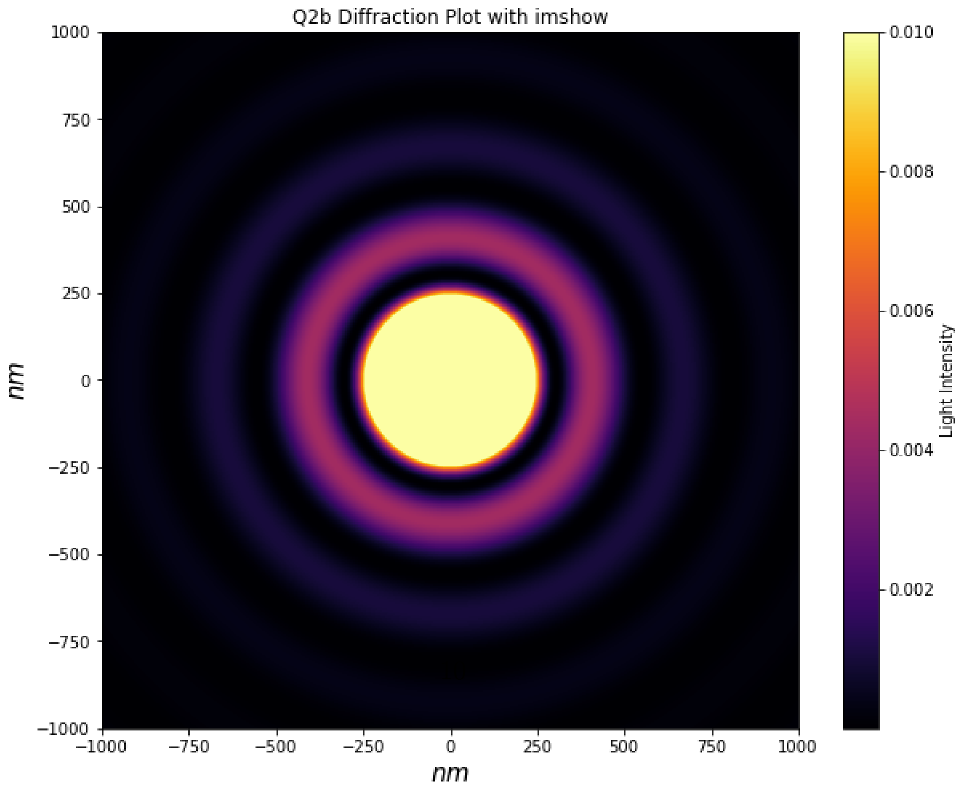
And I get the following result:

```
Difference between my Bessel and scipy.special.jv with m=0, x=0: -1.1102230246251565e-16
Difference between my Bessel and scipy.special.jv with m=2, x=2: -1.6653345369377348e-16
Difference between my Bessel and scipy.special.jv with m=1, x=2.5: 3.885780586188048e-16
```

The difference is at $\mathcal{O}(10^{-15})$! Which is the accuracy limit of the computer. I can conclude that `scipy.special.jv` is using a similar algorithm as my Bessel function.

2.2.3 Exercise 5.4(b)





So I plot the diffraction pattern using both `matplotlib.pyplot.pcolormesh()` and `matplotlib.pyplot.imshow()` on a 2000nm by 2000nm range.
According to Hint 1, I preset the center intensity $I(r = 0)$ at 0.25 manually, since computer program is not good at dealing with a zero on denominator. Also, according to Hint 2, I set `vmax = 0.01` in both `matplotlib.pyplot.pcolormesh()` and `matplotlib.pyplot.imshow()`, in order to achieve the best sensitivity for outer rings. Since the center spot goes up to 0.25, while the outer rings' intensity is at the level of 10^{-3} to 10^{-4}.
They are basically the same plot. I just chose a different color-map for `imshow()` because its name 'inferno' is very cool. And it does look cool indeed!

3 Question 3

3.1 Q3c

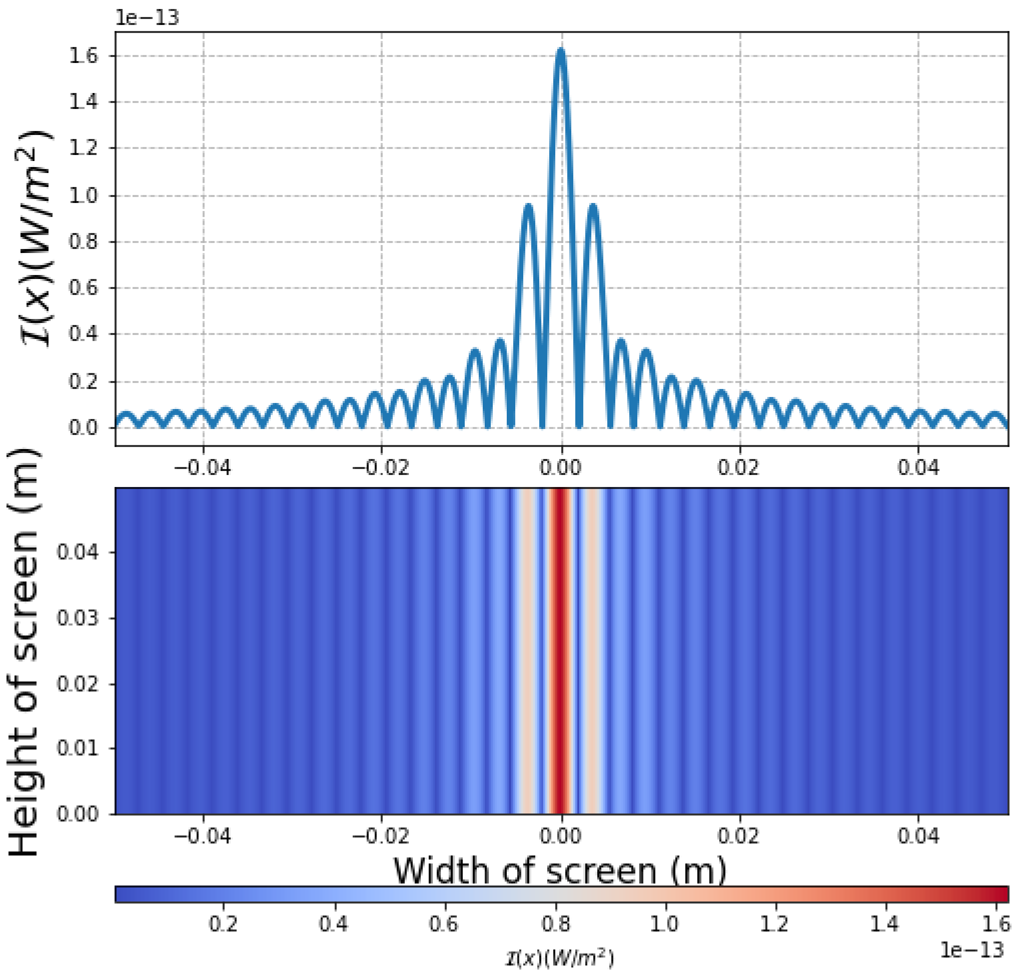
```
# Pseudo code starts here
# define function to do the integral for intensity calculation
f = 1cm # focal length
lambda = 500nm # wavelength
def I(x):
    I = 0 # initialize
    du = step size   # define step size for the integral
    u = array   # array of location on the diffraction grating
    N = u.shape   # N is size of integral
    # define bounding points for Simpon's rule
    a = u[0]
    b = u[-1]
    # Simpson's rule starts here
    I = np.sqrt(a)*np.exp(2j*pi*a*x/lambda/f) + np.sqrt(b)*np.exp(2j*pi*b*x/lambda/f)
    for k in range(1, N/2):
        unow = a + (2*k - 1)*du
        I += 4*np.sqrt(unow)*np.exp(2j*pi*unow*x/lambda/f)
    for k in range(1, N/2):
        unow = a + (2*k)*du
        I += 2*np.sqrt(unow)*np.exp(2j*pi*unow*x/lambda/f)
    I = np.abs(h/3*I)**2
    return I

x = some array # define array for x
Ix = some array # define array for intensity I
for i in range(x.shape[0]):
    Ix[i] = I(x[i]) # calculate intensity for this x
```

I choose to use Simpson's rule when calculating the integral. The reason for that is because the transmission function and the integrand are both highly non-linear functions. Thus Simpson's rule will outperform the Trapezoid rule, as the latter one would have larger truncation errors. I choose the step size to be slightly smaller than *alpha*. I would like it to be small because I want to ensure the accuracy of integral. However, I do not decrease it to be too small, in order to avoid too much computational cost.

3.2 Q3d

3.2.1 Plot

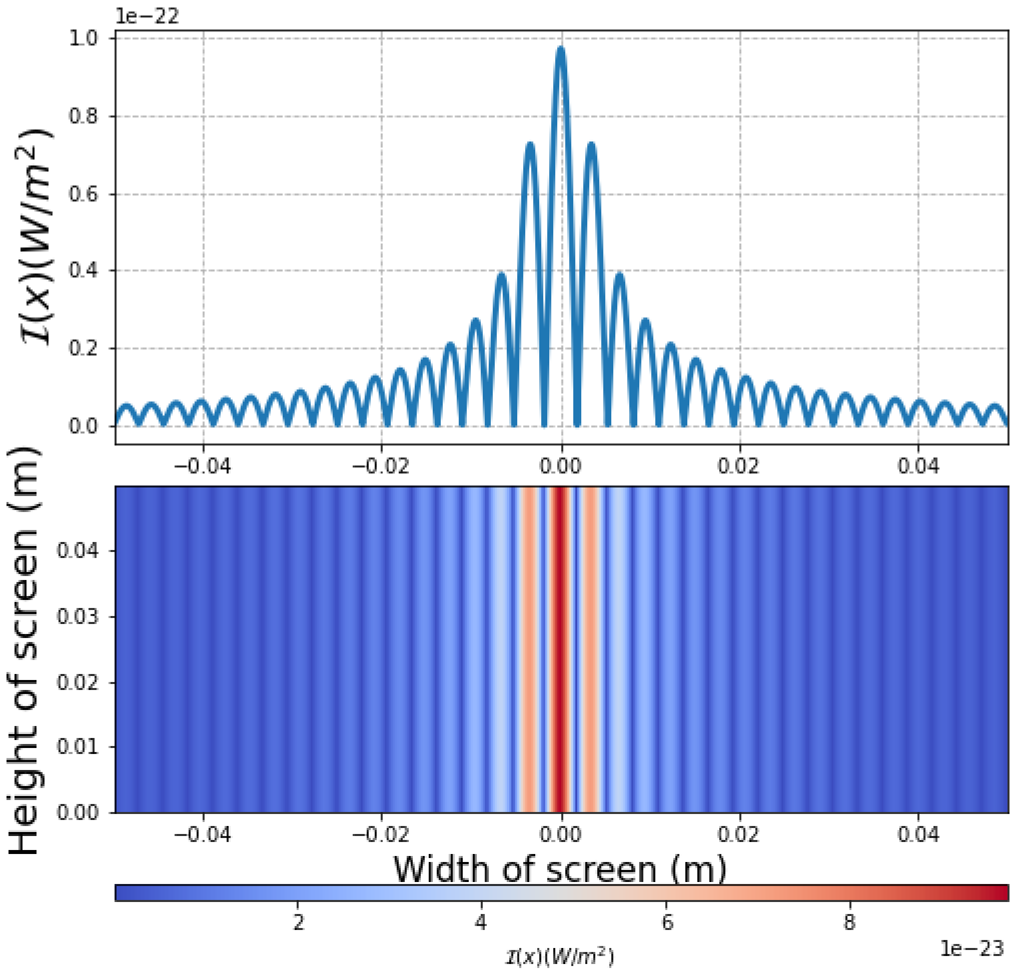


3.2.2 Comments

The figure of intensity $I(x)$ and the density map are plotted above. You can see that we have a central maximum, and minor maxima with gradually decreasing magnitudes lying alongside.

3.3 Q3e

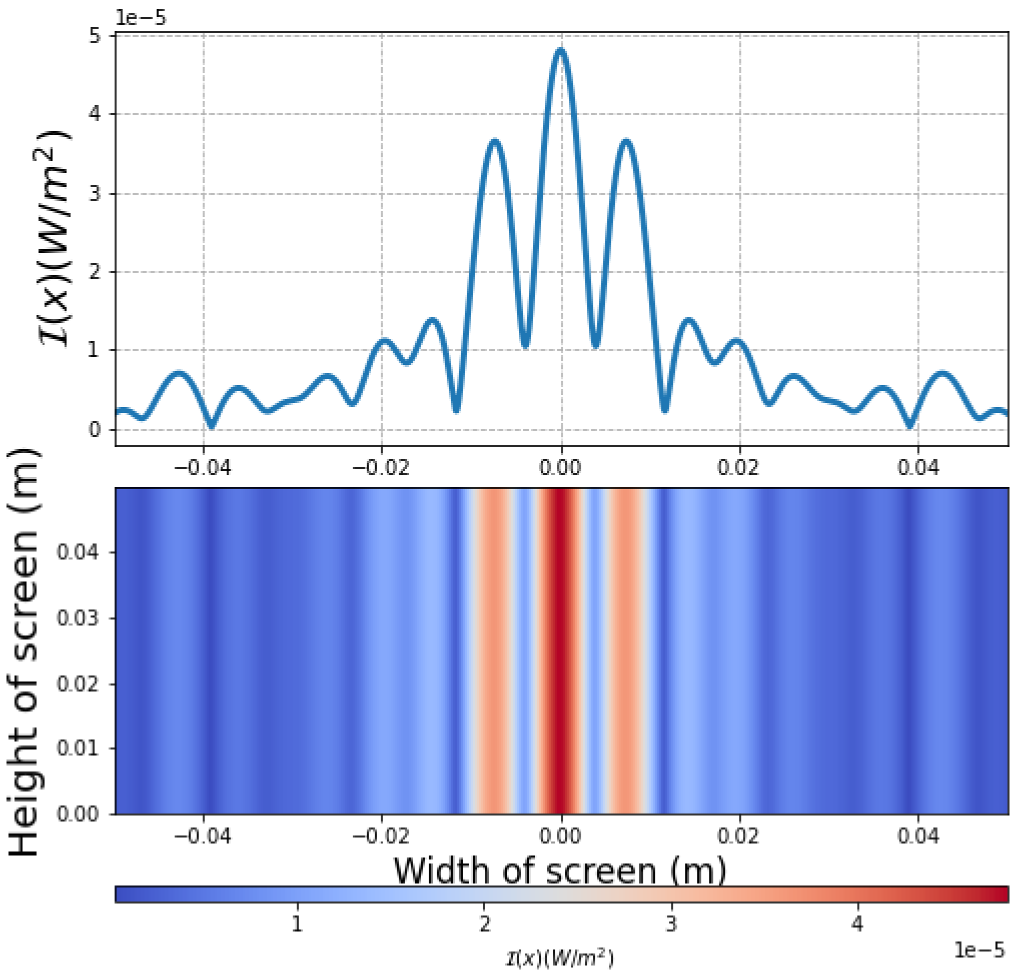
3.3.1 Plot for (i)



3.3.2 Comments

$I(x)$ and the density map are plotted above. You can see they are very similar to the case with the original transmission function. However, the intensity level is generally much higher and the minor maxima also have larger relative magnitudes.

3.3.3 Plot for (ii)



3.3.4 Comments

$I(x)$ and the density map are plotted above. You can see in the middle there are 3 local maxima. Other minor maxima have asymmetric shapes, which are caused by the non-identical configuration of the "square" slits.