Assignment-3

Shivanshu Shekhar

December 31, 2023

1 Assignments

1. Transform the integral to the dimensionless variable $u = r/a$. You should obtain:

$$I = 2\int_0^1 J_v^2(ku)\, udu + 2\left|\frac{J_v(k)}{K_v(g)}\right|^2 \int_1^\infty K_v^2(gu), udu$$

   Where $k = \kappa a = 2.7$ and $g = \gamma a = 1.2$

2. Graph the integrand in Python from 0 to ∞ (use a semi-log plot, since the function amplitudes vary widely, but the function is positive everywhere). Study its characteristics, since those will be required to design your algorithm. Is the function continuous? Is it smooth everywhere?

3. The integral goes to ∞. So we have to discard part of the integral and numerically compute

$$\int_0^a f(x), dx$$

   What value of a will a good choice? Can you analytically bound

$$\int_0^\infty f(x), dx$$

   for large a? **Hint:** The Bessel functions all have approximations in terms of complex exponentials for large arguments.

4. Use Python's built-in integrator, `quad`, found in `scipy.integrate` to do the integration. Verify the given solution. How many calls were required? (Integrate from 0 to a).

   - You need to define the integrand as a Python function.
   - The function form depends on whether u ¡ 1 or u ¿ 1.

- Ask for full output, and look at `scipy.integrate.quad_explain()` for details. One of the outputs is the number of calls made to the function.

5. Use the trapezoidal method and obtain the integral. How does the error scale with h? See the above example code for how to use the `trapzd` routine in module romberg. Does it matter if $r = 1$ is one of the r_j values or if it lies between some r_j and r_{j+1}?

   - Define a global variable 'count' to keep track of how many calls have beenmade (I discussed global variables in python, in class).
   - Plot the error (defined as difference between the numerical integral and the theoretical value) on the y axis and number of calls on the x axis. What sort of plot should you use?
   - What is the trend? How good is the trapezoidal algorithm?

6. Use the romberg module's qromb to investigate Romberg integration. Integrate the entire integral from 0 to ∞. How does the error scale with number of calls? Plot the trend.

7. Split the romberg integrals into (0, 1) and (1, ∞). Repeat the study. Explain the difference in the number of calls required to achieve a given accuracy.

8. Write a python program that implements qromb using `trapzd` and `polint` from the romberg module.

9. Vary the order of qromb and see how the number of calls scales for a fixed error of 10^{-8}.

10. Use spline integration to improve over trapezoidal integration. What scaling do you get (plot it!)? Can you explain why? Hint: Look at the spline interpolated function near r = 1. Note that simple spline integration is a trivial use of the B-Spline routines in scipy. To get the trivial splines, use

```
import scipy.interpolate as si
tck=si.splrep(x,y) # x,y contains table
```

    This generates cubic splines that correspond to the curve that passes exactly through the points. To interpolate, use

```
yy=si.splev(xx,tck) ## xx contains points at which to interpolate
```

    To integrate, use

```
I=si.splint(a,b,tck)
```

These all effectively use the not-a-knot boundary condition. B-Splines can do much more. They can give *smooth* curves that give more importance to certain points and minimize the least-squares error of the fit.

11. Break up the spline integration into two separate parts, from 0 to 1 and from 1 to u_max. How do the scalings change (plot ...)? What does this tell us about spline fitting and spline integration?

12. Rewrite the python to implement qromb where h is reduced to h/3 on each step and interface it to Python. You cannot use the trapzd routine in module Romberg since it implements a divide by two algorithm. Implement even that routine in python but use polint from the module.Numerically compare its performance with the standard Romberg. Analytically predict the performance difference.

13. Plot all the scalings on a single graph, and compare the various approaches used to understand which method works best.

2 Romberg Integration

Romberg integration is a numerical technique for approximating the definite integral of a function. It is an extension of the trapezoidal rule and Richardson extrapolation. The method aims to improve the accuracy of the integral approximation by iteratively refining the estimates.

The Romberg integration process involves constructing a table of values, and each entry in the table provides a more accurate estimate of the integral. The entries are computed using a combination of trapezoidal rule results with different step sizes.

Let's denote the integral we want to approximate as $I = \int_a^b f(x)\,dx$, where $f(x)$ is a continuous function over the interval $[a, b]$.

The trapezoidal rule is given by:

$$T(h) = \frac{h}{2}\left[f(a) + 2\sum_{i=1}^{n-1} f(a+ih) + f(b)\right]$$

where h is the step size, given by $h = \frac{b-a}{2^n}$, and n is the number of subdivisions.

The Romberg integration process involves creating a table of approximations. The table is populated using the trapezoidal rule with varying step sizes. The general formula for the entries in the Romberg table is:

$$R_{i,j} = \frac{4^j R_{i,j-1} - R_{i-1,j-1}}{4^j - 1}$$

where $R_{i,j}$ represents the approximation at the i-th row and j-th column of the Romberg table.

The Romberg integration algorithm can be summarized as follows:

1. **Initialize the table:**
$$R_{0,0} = T(h_0)$$
where $h_0 = \frac{b-a}{2}$ is the initial step size.

2. **Iterate to fill the table:**
$$R_{i,j} = \frac{4^j R_{i,j-1} - R_{i-1,j-1}}{4^j - 1}$$
for $j = 1, 2, \ldots, i$, where i is the current row number.

3. **Repeat until desired accuracy is achieved.**

The final approximation for the integral is given by the last entry in the first column of the table: $R_{k,0}$, where k is the final row number.

The Romberg integration method provides a more accurate result compared to the trapezoidal rule alone, making it a valuable tool for numerical integration when high precision is required.

3 Solutions

3.1 Transformation using the dimensionless variable $u = r/a$

$u = r/a$ means $r = a \cdot u$ and $dr = a \cdot du$ and if $r = 0 \implies u = 0$ and $r = a \implies u = 1$

On substituting we get,

$$I = 2\int_0^1 J_v^2(ku)\, udu + 2\left|\frac{J_v(k)}{K_v(g)}\right|^2 \int_1^\infty K_v^2(gu), udu$$

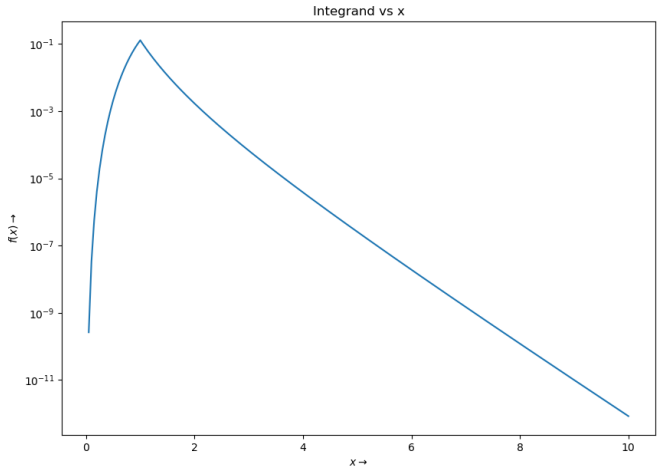
3.2 Plot the integrand in python

We can see that the integrated consists of two parts:

$$f(u) = 2J_v^2(ku)u, \text{ for } 0 \leq u < \infty \tag{1}$$

$$f(u) = 2\left|\frac{J_v(k)}{K_v(g)}\right|^2 K_v^2(gu)u, \text{ otherwise} \tag{2}$$

We plot the integrand below:



We can see from the linear plot that the function reaches its maximum value of nearly 0.13 at point $x = 1$, but this is not visible in the logspace graph as 1 doesnt even exist on logspace, but if we were to increase the sampling then we could possibly see this point. The function is not smooth but continuous at x=1 since it has a very sharp variation in the second derivative, forming a discontinuity. This will require a very small value of h when using trapzd integration.

3.3 Analytically evaluating upper bound of integral

For $r >> v$, $K(2)$ approximately becomes $\frac{\pi}{\sqrt{2\pi x}}e^{-x}$. So we have:

$$\begin{aligned}
\int_a^\infty f(x)dx &= 2|\frac{J_v(k)}{K_v(g)}|^2 \int_a^\infty K_v^2(gu)udu \\
&= 2|\frac{J_3(2.7)}{K_3(1.2)}|^2 \int_a^\infty K_3^2(1.2u)udu \\
&= 0.03454 \int_a^\infty e^{-2.4u}du \\
&= 0.01439e^{-2.4a}
\end{aligned}$$

3.4 Using quad for numerical integration

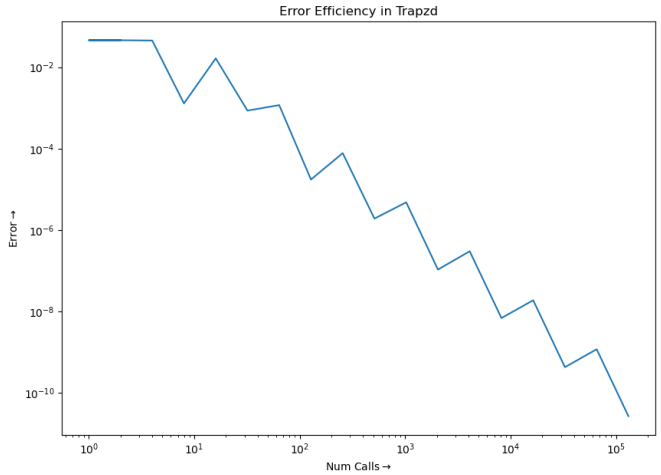
We implement the exact function and next use `scipy.integrate.quad` to integrate the integrand function by taking the upper limit as between (5, 20). We plot the error and num of class graphs below:

Below we plot the error vs upper limit plot:

We can see that $x = 15$ is sufficient for getting almost accurate results. We also get a error about 10^{-15} which is within the upper bound calculated in the previous question.

3.5 Using trapzd from the Romberg module

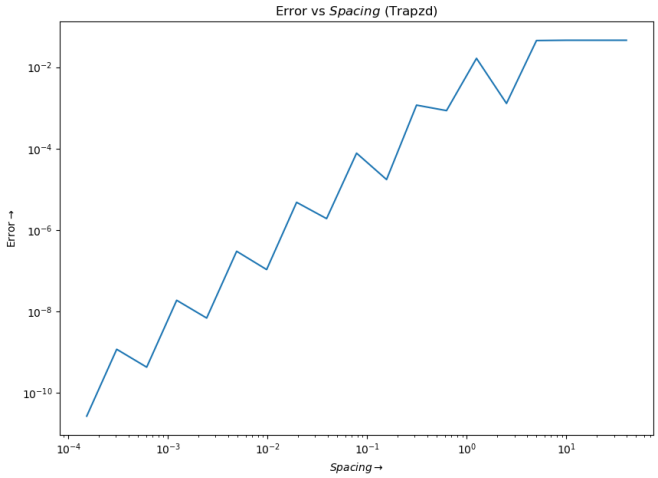
We plot the error in estimation vs the number of function calls below:



From the plot we can infer that the function value is going down with increasing number of function calls. We can also see that trapzd clearly requires **4-5 orders of magnitude more** function calls than quad to obtain

the same level of accuracy. It also matters if $r = 1$ or not, since the function value around the point drops off extremely quickly and hence the integral value will differ.

We also plot the stepsize versus the error in the estimate:

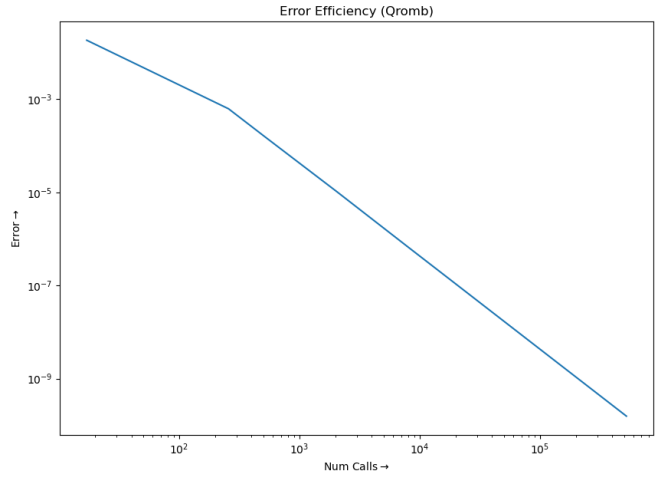


As the step size increases the error increases.

3.6 Using qromb from the Romberg module

If we used qromb for integrating in the interval of (0, 20) with the required tolerance of 10^{-10} gives the number if function calls as N=524289, 3 order of magnitude higher than quad.

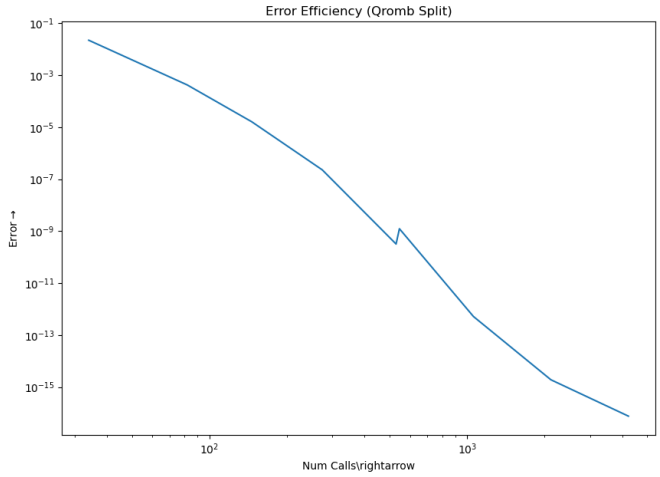
Below we plot the Error vs functional calls plot:



We see that the error drops by an order of magnitude for every magnitude increase in the number of function calls.

3.7 Split integral

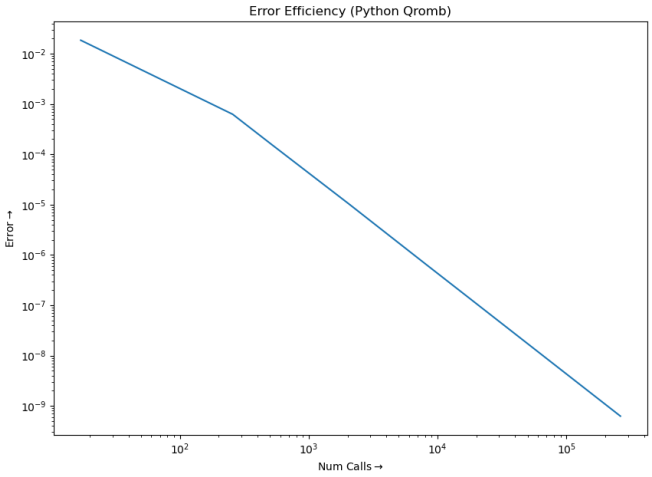
We plot below the error vs number of function calls below:



We can see that the error for the same number of function calls has dropped by 10^{10}, this is intuitive as we could now interpolate better in the non-smooth part of the function.

3.8 Qromb using trapzd and polint

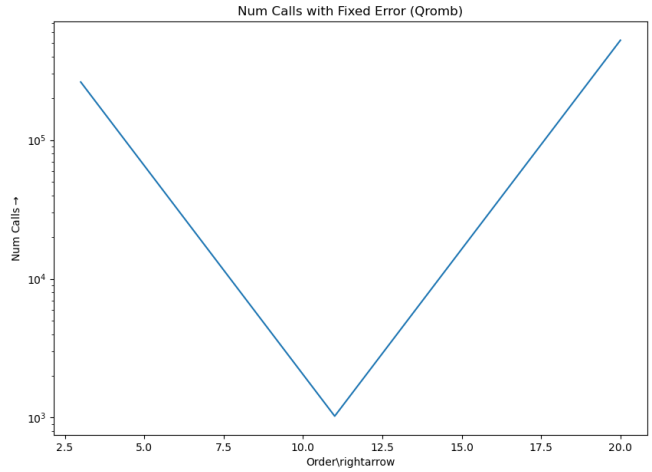
Below we plot the error efficiency of Qromb implemented in python using trapzd and point:



We can see that the error follow the same characteristics as before.

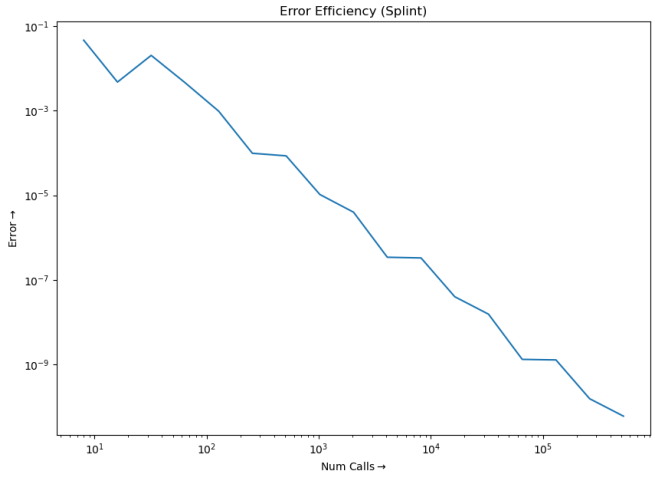
3.9 Changing order

If we fix the error bound to be at 10^{-9} and increase order and plot the number of function calls required to reach this error. We get the following graph

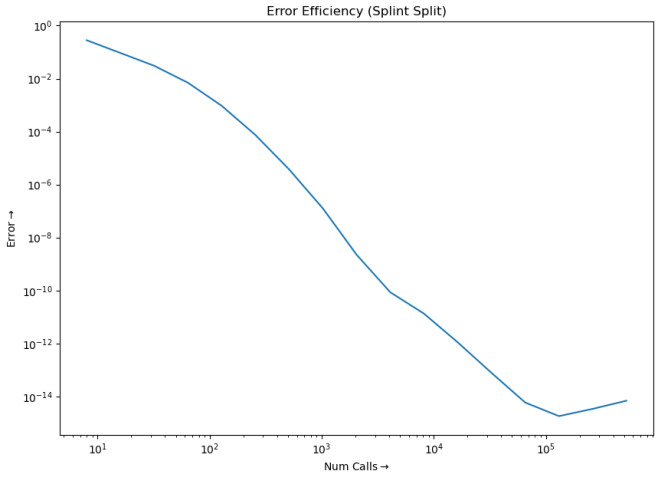


3.10 Integrating using Spline

Using the Spline integration method in scipy we get the following graph:



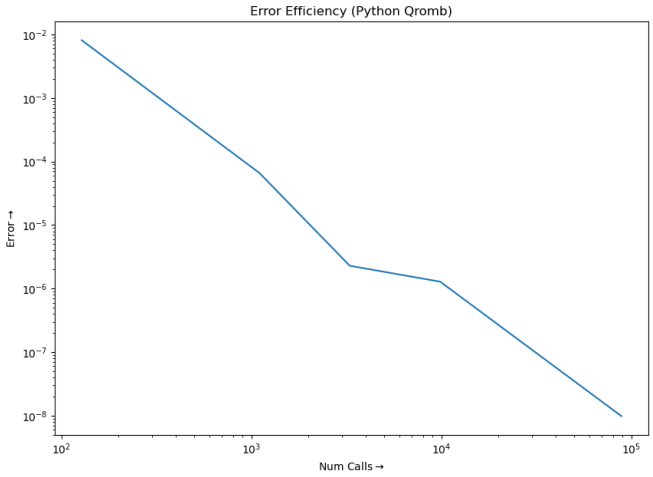
We also split the integral like qromb and we get the following graph for the same:

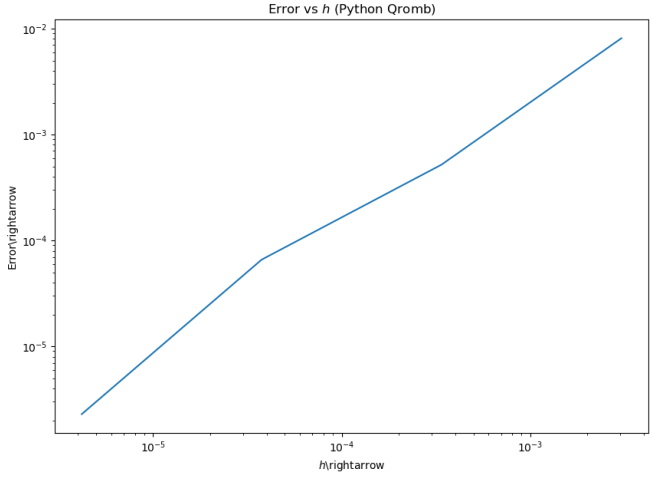


The errors are much lower which can be explained by the fact that spline integration depends on the first derivative being continuous and differentiable and 2nd derivative being continuous.

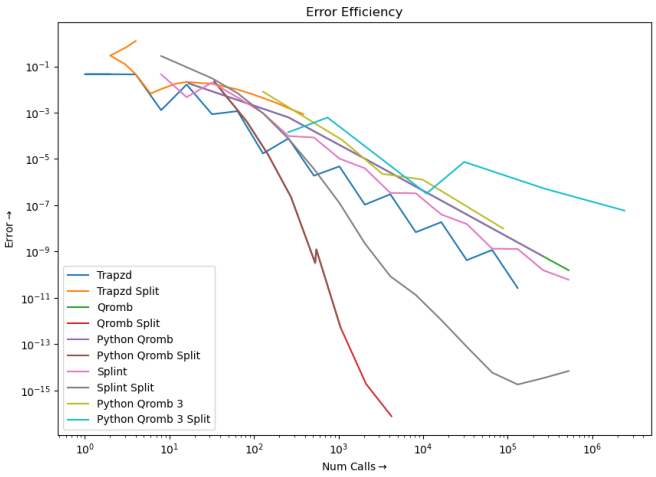
3.11 Divide by 3

Below is the plot for all the divide-by-three algorithms:


Error Efficiency (Python Qromb)
Error →
Num Calls →


Error vs h (Python Qromb)
Error\rightarrow
h\rightarrow

3.12 Plotting All together



From the above graph, it is clear that qromb has the best error efficiency, followed by spline.

4 Conclusion

- We studied the spline integration techniques and saw how the order of interpolation affects the error in the accuracy of the integral.
- We also how the step size is related with the order of error.