#1.

Consider a thin ring with radius r, width dw with uniform surface charge density σ. At a height h above the center of the ring, one can see that by symmetry, the electric field will point straight up. Each element along the loop will contribute to the field in that direction as such:

$$dE_z = |dE|\cos\theta = \frac{\sigma dw \cdot r d\alpha}{4\pi\epsilon_0(r^2+h^2)}\cos\theta = \frac{\sigma dw \cdot r d\alpha}{4\pi\epsilon_0(r^2+h^2)}\cdot\frac{h}{(r^2+h^2)^{1/2}}$$

where θ is the angle between the axis of symmetry of the ring and the infinitesimal element of length $rd\alpha$. To get the total field, we simply integrate over α, which simply gives us a factor of 2π. Hence, the electric field is:

$$\mathbf{E}_{ring} = \frac{\sigma dw \cdot rh}{2\epsilon_0(r^2+h^2)^{3/2}}\hat{\mathbf{z}}$$

Now, for a shell of radius R with surface charge density σ, since we have spherical symmetry, we can align $\hat{\mathbf{z}}$ with the radial direction and integrate over θ for 0 to π. In this setup, $r = R\sin\theta$, $h = z - R\cos\theta$ and $dw = Rd\theta$. Thus, our integral becomes

$$\mathbf{E} = \int_0^\pi \frac{R\sin\theta(z-R\cos\theta)\sigma R d\theta}{2\epsilon_0(R^2-2Rz\cos\theta+z^2)^{3/2}}\hat{\mathbf{r}}$$

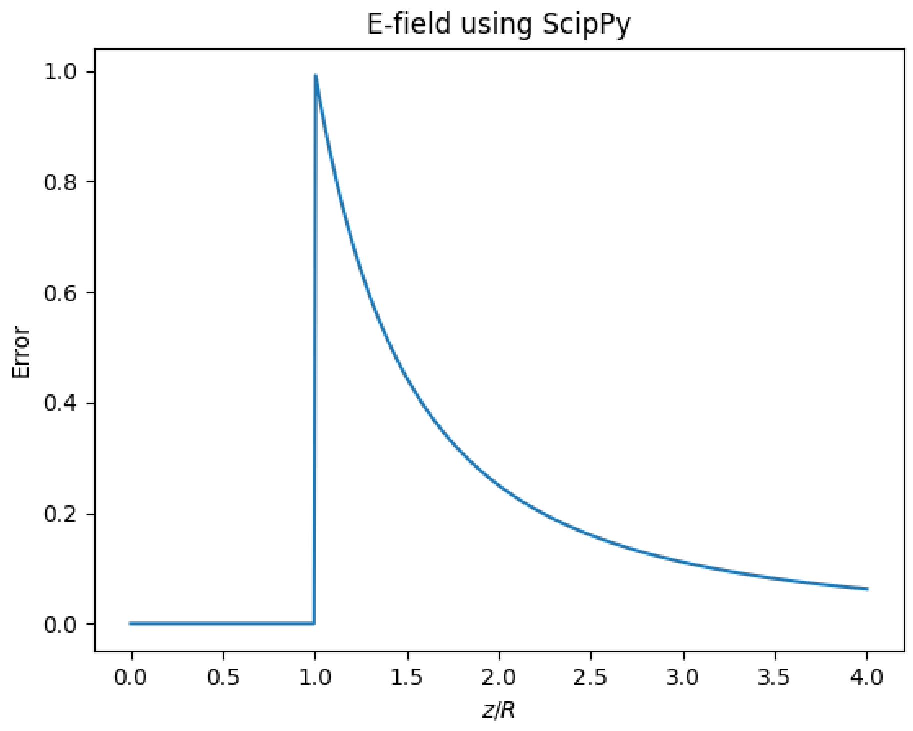
$$= \frac{\sigma R^2}{2\epsilon_0}\int_0^\pi \frac{\sin\theta(z-R\cos\theta)}{(R^2-2Rz\cos\theta+z^2)^{3/2}}d\theta\ \hat{\mathbf{r}}$$

$$= \frac{\sigma}{2\epsilon_0}\int_0^\pi \frac{\sin\theta\left(\left(\frac{z}{R}\right)-\cos\theta\right)}{\left(1-2\left(\frac{z}{R}\right)\cos\theta+\left(\frac{z}{R}\right)^2\right)^{3/2}}d\theta\ \hat{\mathbf{r}}$$

To integrate this, we will first use `scipy.integrate.vec_quad` which is the same as the regular `quad` but it can vectorize the integral on an parameter array (here $\frac{z}{R}$ which goes from 0 to 4). We can compare the output with the analytic result which can easily be obtained via Gauss' Law:

$$\mathbf{E} = \begin{cases} \mathbf{0} & \text{if } z < R \\ \dfrac{\sigma R^2}{\epsilon_0 z^2}\ \hat{\mathbf{r}} & \text{if } z > R \end{cases}$$

Indeed, as seen in Figure 1, the error is pretty much zero, except of course around the discontinuity at $z = R$ where it is on the order of 10^{-12} (still pretty good!).



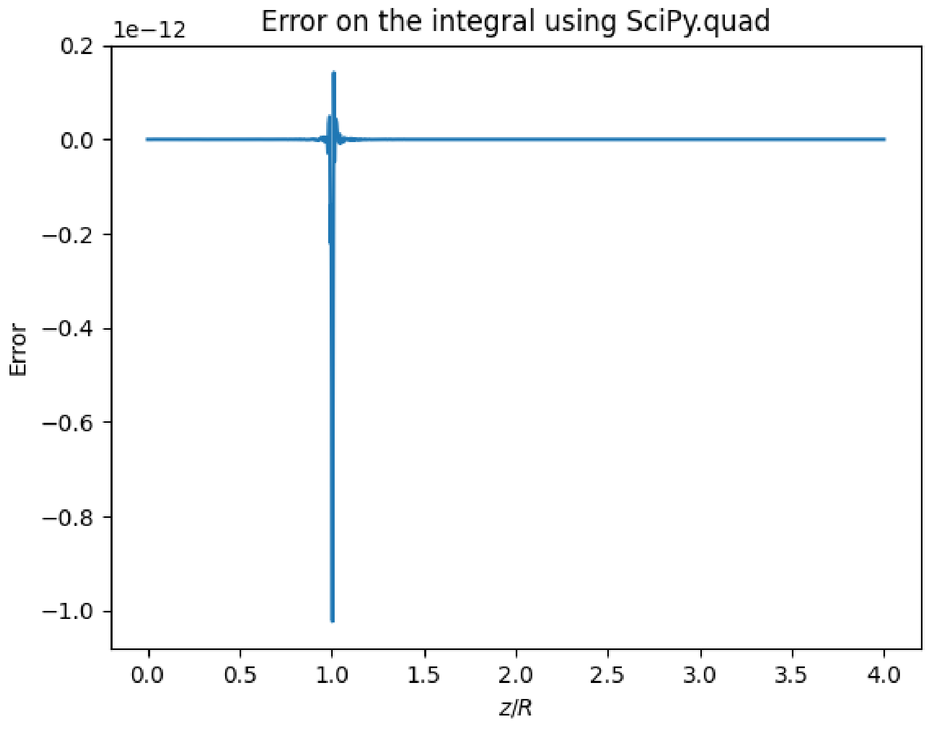


Figure 1: Integral computed using `scipy.integrate.quad_vec`

Next, we will integrate using Simpson's method: we partition $[0,\pi]$ into n equally spaced points θ_i and get

$$\int_0^{\pi} f(\theta)d\theta \approx \left(\frac{d\theta}{3}\right)(f(\theta_0)+4f(\theta_1)+2f(\theta_2)+4f(\theta_3)+2f(\theta_4)+\cdots+f(\theta_{n-1}))$$

Doing this for every value of z, we get an Electric field that also matches the one found analytically. However, there is a divergence at $z = R$ as seen in Figure 2. Away from that problematic point, the error falls to about 10^{-12}.

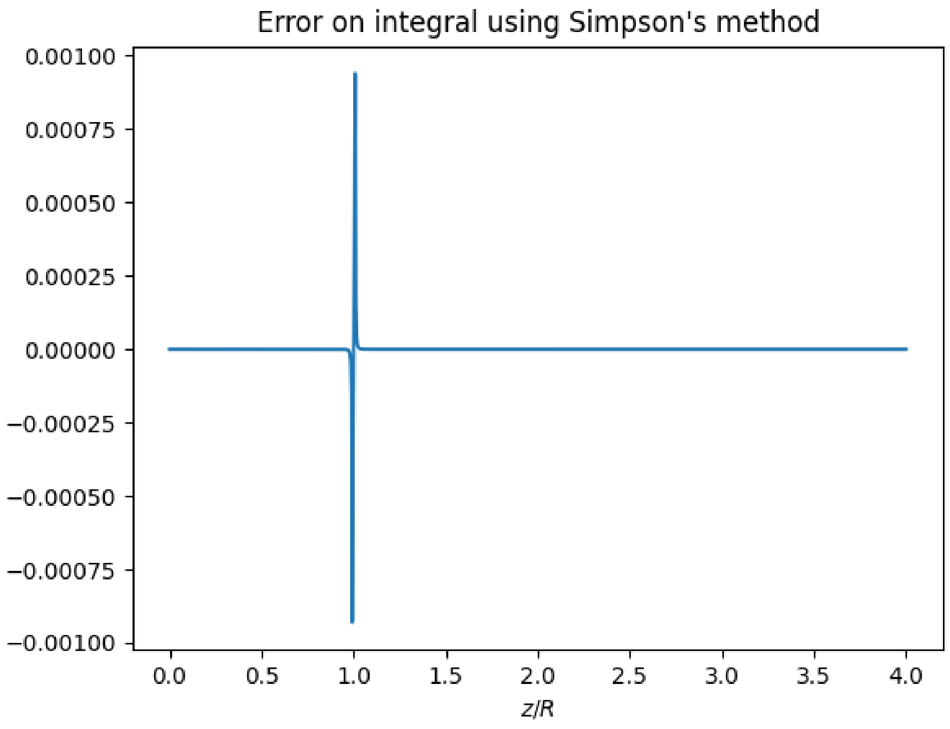


Figure 2: Integral computed using Simpson's method

While Simpson's method wasn't able to deal with this singularity (it returns a `Nan`), Quad seems to handle it without a problem. The value returned at that point by the latter is about 0.5 which is the midpoint between the left and right branches of our result.

#2.

The bulk of the integrator is pretty much the same as seen in class. For a given interval, divide it into 5 points (including the boundary points of said interval) x_0, x_1, x_2, x_3, x_4. Next, compute the integral using Simpson and three points (x_0, x_2 and x_4) as well as with all five points. If the difference between the two is smaller than the input tolerance, return the result. If not, divide the interval into two equal regions and repeat recursively.

To avoid calling the function multiple times for the same x, we can notice that when we divide an interval [x_0,x_4] into [x'_0, x'_4] (left half) and [x''_0, x''_4] (right half), we have a few points in common: $x'_0 = x_0$, $x'_2 = x_1$, $x'_4 = x_2 = x''_0$, $x''_2 = x_3$ and $x''_4 = x_4$ (illustrated in Figure 3). Since we have already evaluated f at these points, there is no need to do so again. We can pass them as an optional argument `extra` and them use them directly.

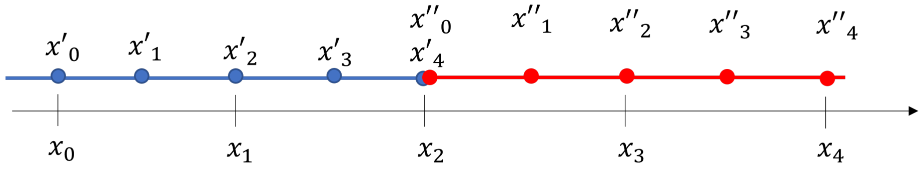


Figure 3: Points in common when dividing into smaller intervals

To verify that indeed no x is passed through f twice, we can keep track by adding said x to an array and a set every time we call $f(x)$. Since sets don't count duplicates, if the number of elements in the list is the same as the number in the set, then we know we didn't call f on the same x multiple times. Using what was described above, we do indeed get that both have the same size. We can also compare the number of function calls to the more naive method by incrementing a variable by 5 each time we call our main function recursively (since we always evaluate the 5 points in the given interval)

Lets try this on a few functions (we set the tolerance to 10^{-10}):

First, $f(x) = \cos(x)$ integrated from -2 to 3 gives `1.0504174348857727` which is about `2e-13` away from the analytical value. We called $f(x)$ 1681 times while the more naive approach calls it 4195 times (a factor of about 2.5).

Now, $f(x) = e^x$ from -2 to 3 gives `19.950201639953367` with an error of about `2e-12`. We called $f(x)$ 2265 times compared to 5655 when using the naive method (again, a factor of about 2.5).

#3.

We start by evaluating $\log_2(x)$ for 40 points between -1 and 1 and fitting a Chebyshev polynomial. Looking at the first few coefficients,

```
[-4.56893391e-01 4.95054621e-01 -4.24689714e-02 4.85763643e-03 -6.25080465e-04
8.57615283e-05 -1.22639505e-05 1.78069535e-06 -2.69126364e-07 3.26169405e-08],
```

we see that in order to have a max error of 10^{-6}, we should keep the first 8 terms and ignore the rest. However, this is for values $x' \in [-1, 1]$. So, to get it for values $x \in [0.5, 1]$ we rescale x using the transformation $x' = 4x - 3$, and *then* evaluate in our fit. Now, to evaluate $\log_2$ for any positive value, we have to first write in in base-2 exponential notation:

$$x = m \cdot 2^p$$

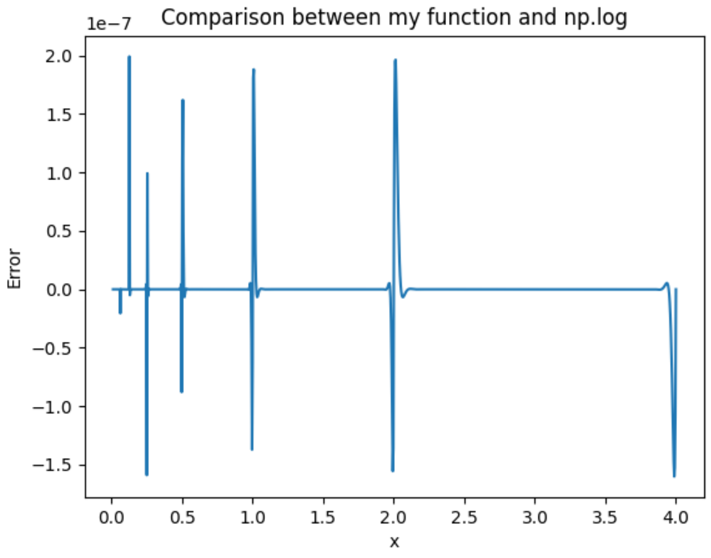
Where m and p are the mantissa and exponent respectively. However, since we assume the mantissa is always normalized and that x is strictly positive, we are assured that $m \in [0.5, 1]$ (if it weren't, multiply/divide by 2 until it is and absorb the extra factors in 2^p). So,

$$\log_2(x) = \log_2(m \cdot 2^p) = \log_2(m) + \log_2(2^p) = \log_2(m) + p$$

To get m and p, we simply use `np.frexp` which serves this exact purpose. Finally, to get the natural log, use the change of base using

$$\ln(x) = \frac{\log_2(x)}{\log_2(e)}$$

Comparing this with NumPy's implementation (see figure below), we get an error of 10^{-7} (away from the spikes, it is around 10^{-15})


Comparison between my function and np.log
1e−7
Error
x