Phys 512 Problem Set 2

Jan Viatteau (260857910)

Problem 1

From a solution to Griffiths 2.7 found on the net (the solution uploaded by Walter Scott on physicsisbeautiful.com), the integral is :

$$\int_0^\pi \frac{(z - R\cos(\theta))\sin(\theta)d\theta}{(R^2 + z^2 - 2Rz\cos(\theta))^{3/2}}$$

once we have gotten rid of any pesky constants that were in front. Assuming $R = 1$, we throw that into the integrator we made for problem 2 (see below), evaluate for a range of z (that includes R) to indeed find that there is a singularity at $z = R$, a divide by zero (we can see that by having $z = R$ and $\theta = \pi/2$. However, `scipy.integrate.quad` doesn't care about the singularity and just spits out the integral. It is probably built strongly enough that if the integral converges, it passes right over the singularity or evaluates it at the closest computable value or something. At any rate, avoiding $z = R$ in our integrator is enough to not have the integrator blow up. There must of course be multiple ways to deal with this better than I have, I simply made it so that at the singularity, don't try to use the integrator and instead use the result computed at the previous point. This causes the single spot where the quad result differs significantly from my integrator's result. The resulting plot is shown below.

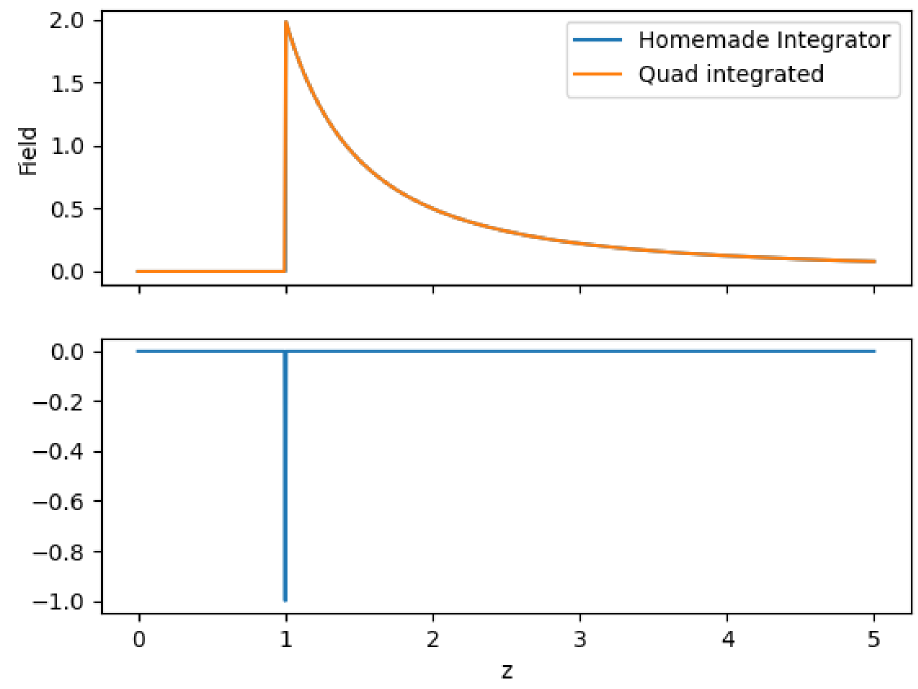

Figure 1: Plot showing the result of the integral using our integrator and quad. As expected, the field is 0 inside the shell (because the sum of the fields from every point on the shell ends up cancelling) and falls off outside the shell (because of the inverse square law). Also, we see that the significant difference between the two integrations lies only at $z = R = 1$, because of the difference of how each handles the integral at singularity.

Problem 2

The idea to make the integrator not compute the $f(x)$ it has already evaluated in earlier calls is to pass that information to sub-calls (when sub-calls are needed). The strategy is this : the initial call has to evaluate the 5 points it needs. If sub-calls are made, they will each have smaller 5 point intervals (one making up the left half of the initial interval, the other the right half). But then, we already evaluated 3 of the 5 points each sub-call needs : the initial endpoints are still endpoints, the initial midpoint is the new endpoint for the sub-calls (right endpoint for the left sub-call, left endpoint for the right sub-call), and the initial 2nd and 4th points are the midpoints for the left and right sub-calls, respectively. Then, we only need to evaluate the 2nd and 4th points for each sub-call. The result is that we avoided doing 3 evaluations when doing a sub-call, so depending on the complexity of the function to integrate and the amount of sub-calls we have to do, we can do less than half the number of evaluations needed and therefore save a lot of computing power and time. Below are 3 examples (of integrals of sine, exp and log between $x = 0.5$ and 5), with the result and number of evaluations saved thanks to passing information to sub-calls.

```
The integral of <ufunc 'sin'> is 0.5939203803871616 computed with
180 evaluations saved
The integral of <ufunc 'exp'> is 146.76443784957326 computed with
522 evaluations saved
The integral of <ufunc 'log'> is 3.8937631325453412 computed with
192 evaluations saved
```

Problem 3

First, to get our model of log base 2, we get a bunch (100) of x/y points between 0.5 and 1 to start off our model. Then, we rescale the points so that the x range falls within the $[-1, 1]$ interval (basically instead of directly using our x array, we will be using a $4x - 3$ array that falls within $[-1, 1]$). We then feed the points into `np.polynomial.chebyshev.chebfit` to get the coefficients for a 50 order Chebyshev model. Looking at the coefficients using a for loop, we truncate the high order coefficients that are lower than the max error we want, 10^{-6}. Then, inside the mylog2 routine, we separate the input number into its mantissa m and exponent e using `np.frexp`. The idea is that the mantissa will always be between 0.5 and 1 (as long as we do not input a negative number, but trying to take the logarithm of a negative number can only end in tragedy anyway), so we can use our Chebyshev model to determine the log base 2 of m (not forgetting to rescale it).

$$\begin{aligned} x &= m \times 2^e \\ \log_2(x) &= \log_2(m \times 2^e) \\ &= \log_2(m) + e \end{aligned}$$

Then, to get the total log base 2 of the input, we simply add the untouched exponent and output. To get the natural log, we simply multiply the result by $ln(2)$, as stated by the change of basis formula. The result of our routine is plotted below, between 0.1 and 10.

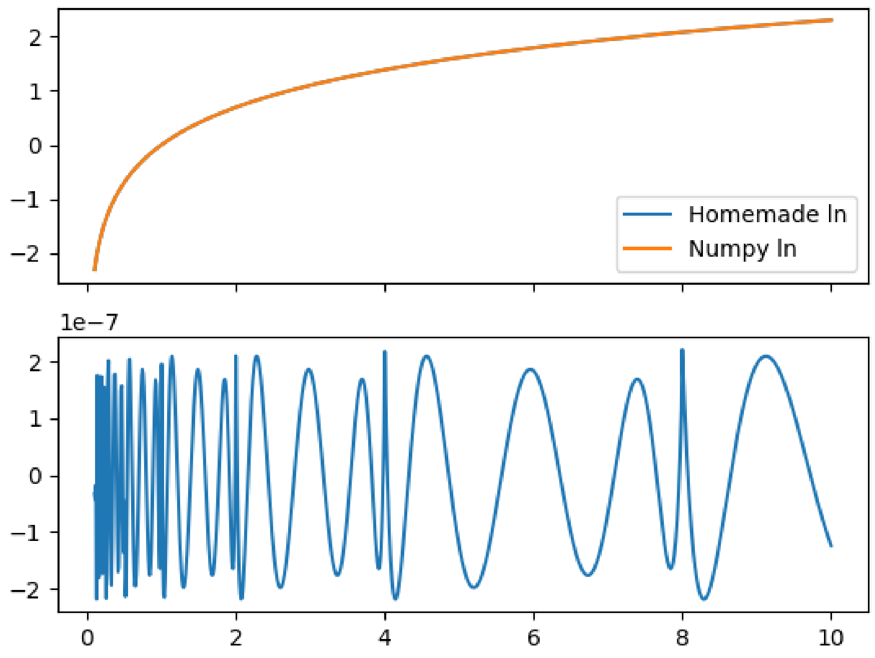

Figure 2: Our homemade, Chebyshev model for the natural logarithm plotted with the numpy natural logarithm, and the residuals between them. As we can see, the two are very close to each other, and the residuals are consistently below our max accepted error of 10^{-6}