PHYS-512: Problem Set 2

Simon Briesenick

September 24, 2022

1. The electric field at a distance z from the sphere's center in the presence of a charged spherical shell of radius R with total charge Q is obtained by performing the integral

$$\mathrm{E}(z) = \int_{\phi=0}^{2\pi} \int_{\theta=0}^{\pi} k \frac{\sigma R^2 \sin\theta (z - R\cos\theta)}{(R^2 + z^2 - 2Rz\cos\theta)^{(3/2)}} \mathrm{d}\theta \mathrm{d}\phi \tag{1}$$

with $\mathbf{E} = \mathrm{E}(z)\hat{\mathbf{e}}_\mathbf{z}$, $k = 1/(4\pi\epsilon_0)$ and ϕ being the azimuth angle. A few simplifications lead to the final expression

$$\mathrm{E}(z) = 2\pi k\sigma R^2 \int_{u=-1}^{1} \frac{(z - Ru)}{(R^2 + z^2 - 2Rzu)^{(3/2)}} \mathrm{d}u. \tag{2}$$

with $u = \cos\theta$ and $\mathrm{d}u = -\sin\theta \mathrm{d}\theta$. The resulting field is shown in Fig. 1. The numerical integrator (succeeding problem) gives the same result as `scipy` - at the singularity, the integrator evaluates the function to `NaN`. In theory, the plot can be improved by masking the integral data to exclude the `NaN`, so that missing data points are still connected by `pyplot`. I decided against that to emphasize that the numeric integrator cannot handle (integrable) singularities.

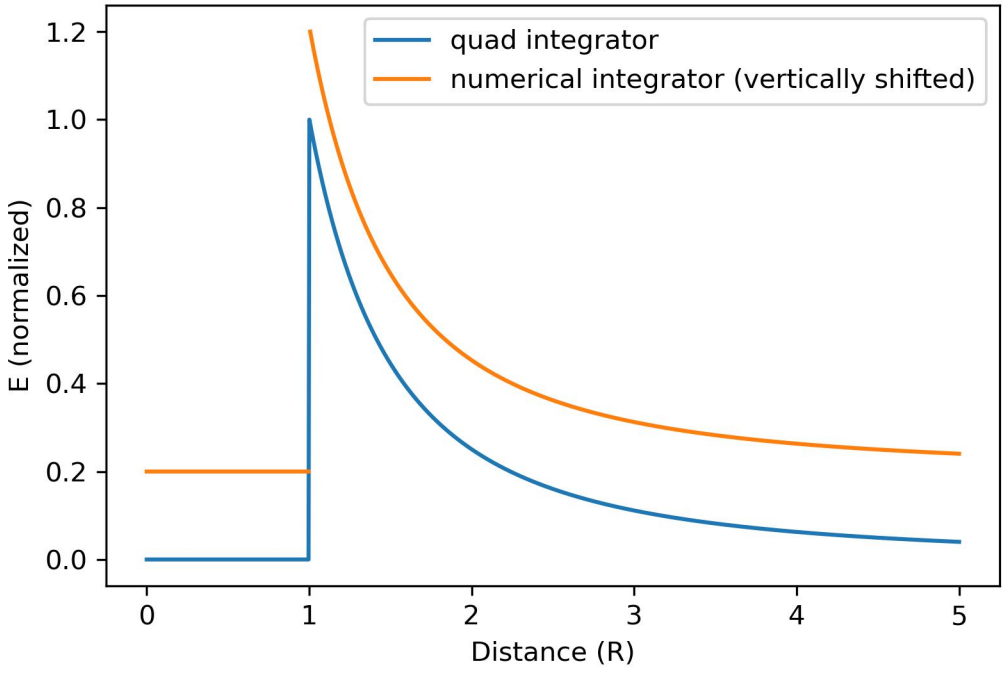

Figure 1: Electric field of a charged, spherical shell as a function of distance from the midpoint (in units of radii R).

2. As an improved version to the numerical integrator that was written in class,

```
integrate_adaptive(fun, a, b, tol, args, extra)
```

calls the input function `fun` for given `x` only under the condition that this value has not previously been calculated. This is done with the help of the variable `extra` which is initially set to `None`. In the function body, `extra` is then defined as an empty dictionary that stores the `x` values as keys and the corresponding function values as values. The function is only called when the `x` value for which it is called is not in `extra.keys()`.

As it is written, the integrator can raise two kinds of `Error`:

(a) The `ZeroDivisionError` is raised when the function is evaluated at a singularity.

(b) A `SpacingTooSmallError` is raised when the `dx` spacing gets smaller then machine precision. This can also happen when a singularity is present in the domain without a function evaluation directly at the singularity. This causes the difference between the three and five point integral to be divergent - subdividing the domain into smaller and smaller regions.

In comparison with the integrator written in class, the adaptive integrator calls the function less times, depending on the how small `tol` is chosen. For a few calls of some typical functions:

```
>>> integrate_adaptive(lambda x, args=None: x**2, 0, 1, tol=1e-5)
Integral of <lambda> from  0 to 1:  0.3333333333333333
number of fun calls (lazy):  5
number of fun calls (adaptive):  5
>>> integrate_adaptive(np.exp, 0, 1, 1e-5)
Integral of exp from  0 to 1:  1.7182819740518918
number of fun calls (lazy):  35
number of fun calls (adaptive):  17
>>> integrate_adaptive(np.sin, 0, np.pi/2, tol=1e-5)
Integral of sin from  0 to 1.57:  1.0000001741482483
number of fun calls (lazy):  55
number of fun calls (adaptive):  25
>>> integrate_adaptive(np.sin, 0, np.pi/2, tol=1e-10)
Integral of sin from  0 to 1.57:  1.000000000001764
number of fun calls (lazy):  1035
number of fun calls (adaptive):  447
>>> integrate_adaptive(np.sin, 0, np.pi/2, tol=1e-15)
Integral of sin from  0 to 1.57:  1.000000000001764
number of fun calls (lazy):  17925
number of fun calls (adaptive):  7553
```

The last three examples show the number of function calls with both methods as `tol` is decreased. It's notable, that the ratio of `number of fun calls (lazy)` with `number of fun calls (adaptive)` approaches a limit of 2.5 as `tol` approaches 0. This is understood by the fact, that for each recursive call, the integrator we wrote in class calls the function 10 times, while the adaptive version calls it only 4 times.

3. To generate the data to be fit, `np.log2` was called on 1000 values for `x`. To fit a chebyshev expansion to the data, I fit the generated `y` values on a separate variable, `u`, that is related to `x` by $u(x) = 4x - 3$. This way, the domain will be bound between values of ± 1. A high order expansion (`ord = 50`) gives the following coefficients for the first 50 chebyshev polynomials

```
   [-4.56893394e-01  4.95054673e-01 -4.24689768e-02  4.85768297e-03
 -6.25084976e-04  8.57981013e-05 -1.22671891e-05  1.80404306e-06
-2.70834249e-07  4.13047215e-08 -6.37809322e-09  9.94825980e-10
-1.56462339e-10  2.47803550e-11 -3.94889258e-12  6.33002244e-13
-1.02508515e-13  1.72168063e-14 -3.28415609e-15  1.15259259e-15
-7.44053019e-16  6.57825532e-16 -9.12606686e-16  5.63778404e-16
-8.97638568e-16  6.31831384e-16 -9.62020296e-16  5.66468286e-16
-5.05759348e-16  6.03335064e-16 -4.26387756e-16  6.87192004e-16
-6.54400968e-16  6.39689764e-16 -7.55970615e-16  3.64402184e-16
-7.18736672e-16  5.53505805e-16 -4.49797770e-16  5.22170147e-16
-4.38617058e-16  5.49127498e-16 -3.49386688e-16  4.54497703e-16
-6.04583404e-16  3.73461271e-16 -2.15560790e-16  1.87622290e-16
-3.41916049e-16  1.15900577e-16 -9.29112046e-17]
```

As expected, the absolute values of the coefficients decreases with higher order and are indicative of the maximum error in the fit. For an accuracy better than 10^{-6}, one hence needs to retain the expansion up until the ninth order term. A comparison between the modelled data and the calculated data is shown in Fig. 2. To calculate the $\log_2$ of an arbitrary number, I first used `np.frexp` to separate the input variable to

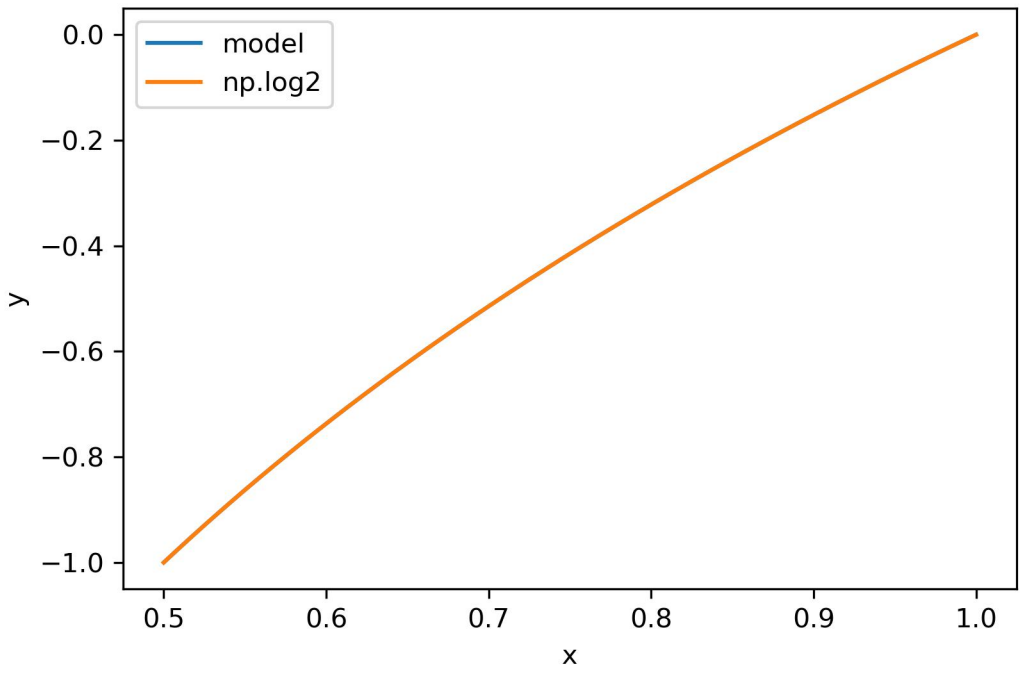

Figure 2: Comparison of the function values with the modeled data.

`mylog2(x)` into a pair of numbers, `(mantissa, exponent)` such that `x = mantissa*2**exponent`. With the input value rewritten in this form, the $\log_2(x)$ is easily calculated using the `coeffs` array from part (a). The variable `mantissa` needs to be treated exactly as `x` in the previous part:

```
def mylog2(x):
    (mantissa, exponent) = np.frexp(x)
    variable = 4*mantissa - 3
    cheb = np.empty(10)
    cheb[0]=1
    cheb[1]=variable
    for i in range(1, 9):
        cheb[i+1] = 2*variable*cheb[i] - cheb[i-1]
     model = coeffs[0:10]*cheb
    term1 = sum(model)
    return term1 + exponent
```

Some test cases:

```
>>> mylog2(2)
1.0000000075588507
>>> mylog2(0.5)
-0.9999999924411493
>>> mylog2(4)
2.0000000075588504
```