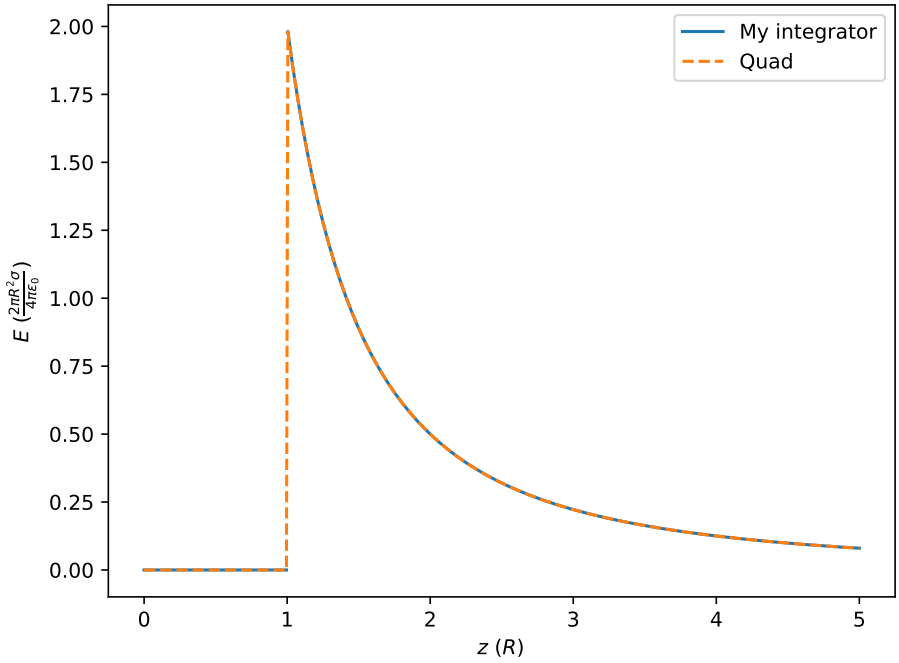

Figure 1: Plot of E field of spherical shell as a function z using the recursive variable step size integrator from problem 2 and `scipy.integrate.quad`. The z axis is in unit of the sphere's radius R and E axis is in unit of $\frac{2\pi R^2\sigma}{4\pi\epsilon_0}$.

1. The integral is

$$\frac{2\pi R^2\sigma}{4\pi\epsilon_0}\int_{-1}^{1}\frac{z-Ru}{(R^2+z^2-2Rzu)^{3/2}}\,\mathrm{d}u\,.$$

   We used the integrator from problem 2. The plot of the E field as a function of z is shown in Figure 1. There is a singularity of the integrand when $z = R$. Our integrator cannot handle the case of $z = R$ where the integrand diverges while `scipy.integrate.quad` can. In my Python implementation, I manually checked for poles when doing the integration and returns `None` if a pole is detected.

2. The Python implementation of my integrator is as follows

```
import numpy as np

import sys
sys.path.append("../")
from utils.logger import Log


# The following code is taken/adapted from integrate_adaptive_class.py
def integrate(fun,a,b,tol):
    global lazycounter

    x=np.linspace(a,b,5)
    dx=x[1]-x[0]
    y=fun(x)
    # Call fun 5 times
    lazycounter += 5
```

```
    #do the 3-point integral
    i1=(y[0]+4*y[2]+y[4])/3*(2*dx)
    i2=(y[0]+4*y[1]+2*y[2]+4*y[3]+y[4])/3*dx
    myerr=np.abs(i1-i2)
    if myerr<tol:
        return i2
    else:
        mid=(a+b)/2
        int1=integrate(fun,a,mid,tol/2)
        int2=integrate(fun,mid,b,tol/2)
        return int1+int2
# Copied code end here


def integrate_adaptive(fun, a, b, tol, extra=None):
    """
    My integrator with adaptive step size, with a function calling counter
    :param fun: integrand
    :param a: lower bound
    :param b: upper bound
    :param tol: tolerance
    :param extra: pre-computed data
    :return: integration result
    """
    # Conuter for counting function calls
    global mycounter

    # 5 evenly spaced points
    x = np.linspace(a, b, num=5)

    # Handle poles
    try:
        # Try-except can only capture error
        with np.errstate(invalid='raise'):
            # If precomputed value is not passed, we computed all five points
            if extra is None:
                y = fun(x)
                # Need to call fun 5 times
                mycounter += 5
            # Precomputed value for the first, last and the middle points
            else:
                y = np.array([extra[0], fun(x[1]), extra[1], fun(x[3]), extra[2]])
                # Need to call fun 2 times only
                mycounter += 2
    except FloatingPointError:
        # If pole exists, we directly return a None
        print('Invalid value encountered during integration, setting result to None.')
        return None

    dx = x[1] - x[0]

    i1 = (y[0] + 4 * y[2] + y[4]) / 3 * (2 * dx)
    i2 = (y[0] + 4 * y[1] + 2 * y[2] + 4 * y[3] + y[4]) / 3 * dx
```

```
    # Iterative approach
    err = np.abs(i1 - i2)
    if err < tol:
        return i2
    else:
        mid = (a + b) / 2
        # Pass the computed values to our integrator to reduce function calls on fun
        int1 = integrate_adaptive(fun, a, mid, tol / 2, extra=y[:3])
        int2 = integrate_adaptive(fun, mid, b, tol / 2, extra=y[-3:])
        return int1 + int2


def problem2(fun, a, b, tol, name):
    """
    Helper for question 2
    :param fun: integrand
    :param a: lower bound
    :param b: upper bound
    :param tol: tolerance
    :param name: name of the integrand
    :return: None
    """
    global log, mycounter, lazycounter

    # Initialize counter
    mycounter = 0
    lazycounter = 0

    log.append(f'Integrating f(x)={name} between {a} and {b} with max error {tol}:')
    integrate_adaptive(fun, a, b, tol)
    log.append(f'My integrator called f(x) {mycounter} times')
    integrate(fun, a, b, tol)
    log.append(f'Lazy integrator called f(x) {lazycounter} times')


if __name__ == '__main__':
    log = Log()

    # Let's do the computation for f(x)=cos(x) and f(x)=1/(x^2+1)
    problem2(np.exp, 0, 1, 1e-7, 'exp(x)')
    problem2(np.cos, 0, 1, 1e-7, 'cos(x)')
    problem2(lambda x: 1 / (x ** 2 + 1), 0, 1, 1e-7, '1/(x^2+1)')

    log.save('2.txt')
```

For the functions $f(x) = \exp(x)$, $f(x) = \cos(x)$ and $f(x) = \frac{1}{x^2+1}$, the number of calls to $f(x)$ made by my integrator and the lazy one is as follows

```
Integrating f(x)=exp(x) between 0 and 1 with max error 1e-07:
My integrator called f(x) 57 times
Lazy integrator called f(x) 135 times
Integrating f(x)=cos(x) between 0 and 1 with max error 1e-07:
My integrator called f(x) 33 times
Lazy integrator called f(x) 75 times
Integrating f(x)=1/(x^2+1) between 0 and 1 with max error 1e-07:
```

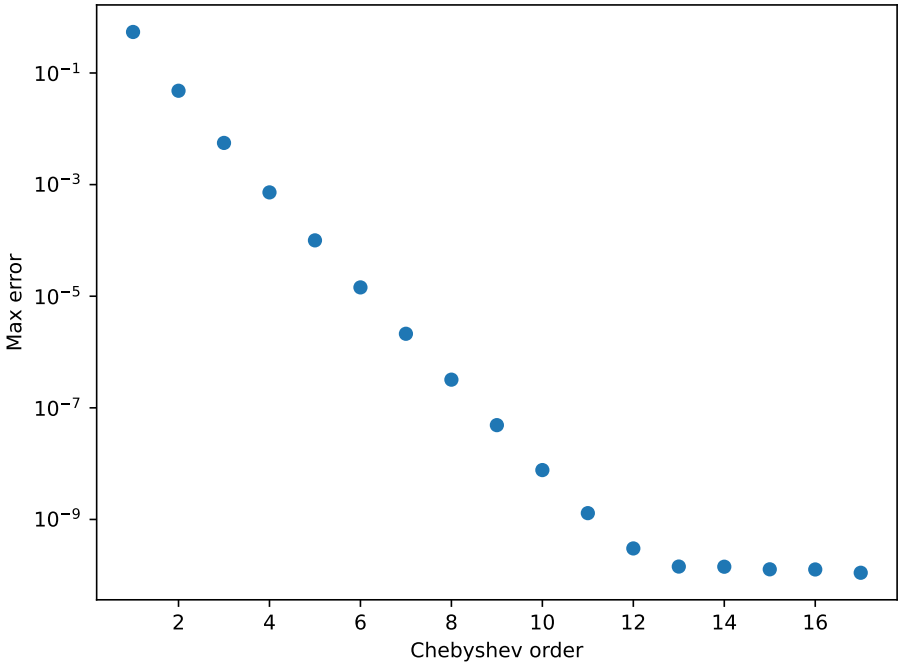

Figure 2: Plot of maximum error of $\log_2$ in $x \in [0.5, 1]$ using different orders of Chebyshev fit.

```
My integrator called f(x) 69 times
Lazy integrator called f(x) 165 times
```

In theory, for a function that requires calling $f(x)$ n times using the lazy implementation, my implementation should call $f(x)$ only

$$n \cdot \frac{2}{5} + 3$$

times. For the first call to my function, it calls $f(x)$ 5 times which is the same as the lazy implementation. But for all subsequent calls to my function in the iteration process, it calls $f(x)$ only 2 times instead of 5 times.

3. I computed the Chebyshev fit to $f(x) = \log_2(x)$ with $x \in [0.5, 1]$ rescaled to $x' \in [-1, 1]$. The error is determined by choosing 50 points in the interval and find the maximum difference to NumPy's $\log_2$. The error becomes less than 10^{-6} when the order is larger than 7 as you can see in Figure 2. My implementation of ln called `mylog2` defaults to using order 8 Chebyshev polynomial.

   I also implemented the fit using both Legendre and Taylor polynomials at the same 8th order. To determine the error associated with each fit method, a log sequence between $10^{0.1}$ to 10^{10} is generated. The RMS and maximum error is computed. The output of my code gives

```
My implementation using Chebyshev has RMS error: 8.03416457185377e-07
My implementation using Chebyshev has max error: 2.559018795977863e-06
My implementation using Legendre has RMS error: 6.658012164212994e-07
My implementation using Legendre has max error: 2.203351474605597e-06
My implementation using Taylor has RMS error: 4.000069556502936
My implementation using Taylor has max error: 13.906620180878882
```

   From the output, we can see that both Chebyshev and Legendre gives error of around the same order of magnitude. Legendre is slightly better than Chebyshev fit. However, error using Taylor series quickly blows up and is significant that the result cannot be trusted.