Assignment 2

Q1.

From solution to Griffith 2.7, we have

$$E_z = \frac{1}{4\pi\varepsilon_0}(2\pi R^2\sigma)\int_0^{\pi}\frac{(z-R\cos\theta)\sin\theta}{(R^2+z^2-2Rz\cos\theta)^{\frac{3}{2}}}\,d\theta$$

let $u=\cos\theta$

$$= \frac{1}{4\pi\varepsilon_0}(2\pi R^2\sigma)\int_{-1}^{1}\frac{z-Ru}{(R^2+z^2-2Rzu)^{\frac{3}{2}}}\,du$$

From this we can set up our integrator.

Assume all coefficients are 1

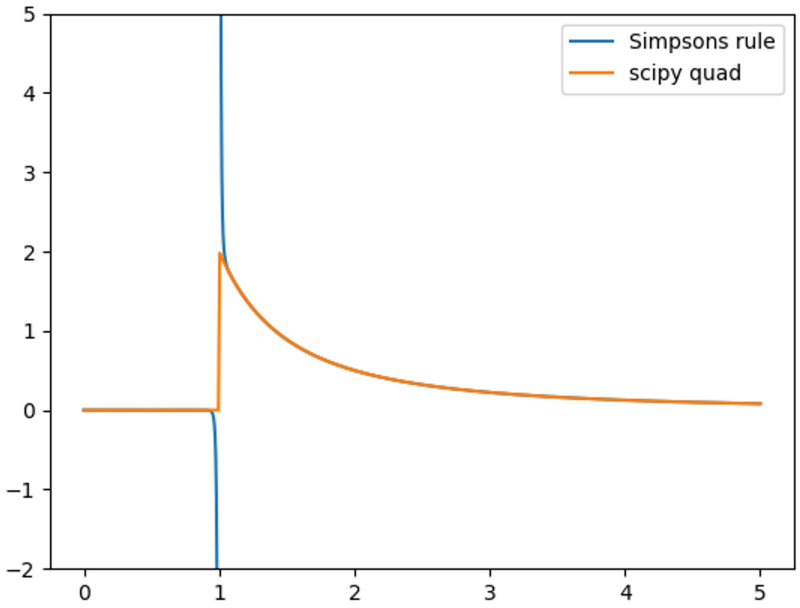
```
import numpy as np
from matplotlib import pyplot as plt
from scipy import integrate
#define function to integrate according to the solution to Griffith 2.7
def fun_int(x,z):
    return (z-x)/((1+(z**2)-(2*z*x))**(3/2))

#define integrator that uses Simpsons rule
def integrator(z):
    x = np.linspace(-1,1,1001) #integrate from -1 to 1
    y = fun_int(x,z)
    dx=np.median(np.diff(x))
    #integrate according to Simpson rules
    ints_simp=dx/3.0*(y[0]+y[-1]+4*np.sum(y[1::2])+2*np.sum(y[2:-1:2]))
    return ints_simp

simp_val = []
quad_val = []
z_arr = np.linspace(0,5,1001)
for z in z_arr: #note z = 1 is included in this linspace
    simp_temp = integrator(z)
    quad_temp,err = integrate.quad(fun_int,-1,1,args=(z,))
    simp_val = np.append(simp_val,simp_temp)
```

```
    quad_val = np.append(quad_val,quad_temp)

plt.clf()
plt.plot(z_arr,simp_val)
plt.plot(z_arr,quad_val)
plt.ylim(-2, 5) #set y-axis limits so that the singularity does not make the plot
look too bad
plt.legend(['Simpsons rule','scipy quad'])
plt.savefig('question_1.png')
plt.show()
```



As we can see, there is a singularity at $z = R$.

Our integrator using Simpsons rule breaks up near the singularity, while scipy.integrate.quad does not care.

Our integrator breaking up is to be

expected, since near the singularity we would have very large value for function value due to a small value dividing a even smaller value, which is pretty bad for computer calculation.

Q2

```
import numpy as np
function_call = 0 #global variable to see how many function calls we did
def exp_int(x):
    global function_call
    #update the total number of function calls
    function_call = function_call+np.size(x)
    return np.exp(x)
def integrate(fun,a,b,tol,extra=None):
    if extra is None:
        x=np.linspace(a,b,5)
        dx=x[1]-x[0]
        y=fun(x)
        #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
            #pass the already calculated function value into recursive call
            int1=integrate(fun,a,mid,tol/2,extra=np.array([y[0], y[1], y[2]]))
            int2=integrate(fun,mid,b,tol/2,extra=np.array([y[2], y[3], y[4]]))
            return int1+int2
    else:
        x=np.linspace(a,b,5)
        #we already have the 1st, middle and last function value from extra
        #need only to calculate 2nd and 4th value
        y=np.array([extra[0],0,extra[1],0,extra[2]])
```

```
        y[1] = fun(x[1])
        y[3] = fun(x[3])
        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
        myerr=np.abs(i1-i2)
        if myerr<tol:
            return i2
        else:
            mid=(a+b)/2
            int1=integrate(fun,a,mid,tol/2,extra=np.array([y[0], y[1], y[2]]))
            int2=integrate(fun,mid,b,tol/2,extra=np.array([y[2], y[3], y[4]]))
            return int1+int2

def integrate_old(fun,a,b,tol):
    x=np.linspace(a,b,5)
    dx=x[1]-x[0]
    y=fun(x)
    #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_old(fun,a,mid,tol/2)
        int2=integrate_old(fun,mid,b,tol/2)
        return int1+int2
ans_new = integrate(exp_int,1,2,1e-6)
print(ans_new)
# ans_old = integrate_old(exp_int,1,2,1e-6)
# print(ans_old)
print(function_call)
```

we can compare our number of function calls by commenting out different part of the last section of our code.

For our new code:

The integrated value is: 4.670774295215381
The number of function calls is: 33

For the old code in class:

```
The integrated value is:  4.670774295215381
The number of function calls is:  75
```

We can see that the value is unchanged, but the number of function calls is drastically reduced.

Q3:

For cheb fit of log2 from 0.5 to 1, we need to renormalize our x value to be

$4x-3$ to have x range from -1 to 1.

```
import numpy as np
from matplotlib import pyplot as plt
x_true = np.linspace(0.5,1,10001) #x array from 0.5 to 1
#renormalize x array to be from -1 to 1 for cheb fit
x_renorm = x_true*4-3
y = np.log2(x_true)
cheb_coeff = np.polynomial.chebyshev.chebfit(x_renorm,y,30)
print(cheb_coeff)
```

the cheb coefficient of degree 30 is

the cheb coefficient of degree 30 is

```
[-4.56893394e-01  4.95054673e-01 -4.24689768e-02  4.85768297e-03
 -6.25084976e-04  8.57981013e-05 -1.22671891e-05  1.80404307e-06
 -2.70834256e-07  4.13047281e-08 -6.37810023e-09  9.94832637e-10
 -1.56469112e-10  2.47866900e-11 -3.95524365e-12  6.38836021e-13
 -1.08832208e-13  2.28575622e-14 -9.50961038e-15  6.55421517e-15
 -6.32919459e-15  5.86956403e-15 -5.59628545e-15  5.11556488e-15
 -5.39073805e-15  4.30425692e-15 -4.66669035e-15  3.68445846e-15
 -4.49188576e-15  2.67766742e-15 -2.39329224e-15]
```

From this we see that to have an error less than 10^{-6}, it is enough to only pick the first 8 coefficients.

```
nuse = 8
cheb_coeff_cut = cheb_coeff[:nuse]
```

For natural log, we could use

numpy.frexp(x) It split a float x into mantissa and exponent such that

$$x = \text{mantissa} \times 2^{\text{exponent}}$$

Note that for positive number, frexp always gives mantissa to be from 0.5 to 1.

Now

$$\log_2(x) = \log_2(\text{mantissa} \times 2^{\text{exponent}})$$

$$= \log_2(\text{mantissa}) + \log_2(2^{\text{exponent}})$$

$$= \log_2(\text{mantissa}) + \text{exponent}.$$

Then we could use

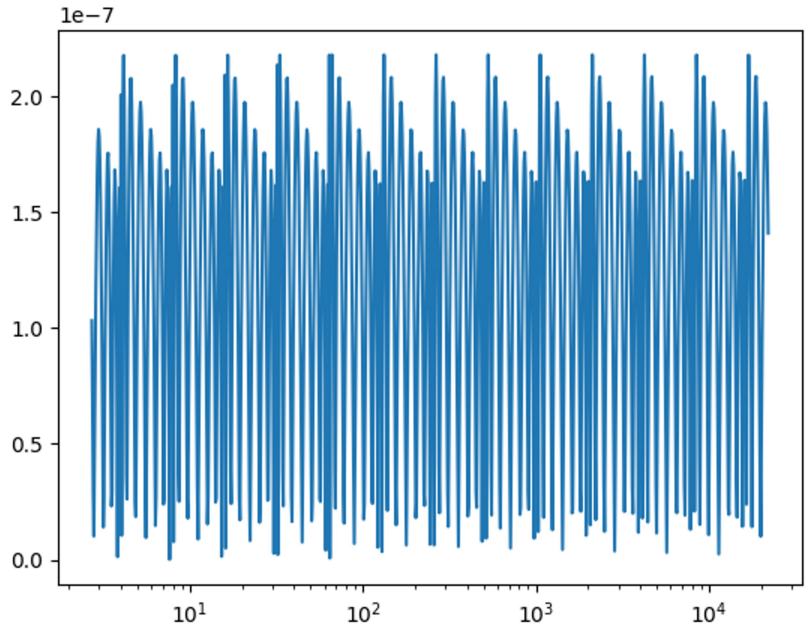
$$\log_e(x) = \frac{\log_2(x)}{\log_2(e)}$$

to get natural log.

```
def mylog2(x):
    #stop calculation if any of the x is smaller or equal to zero
    if np.any(x<=0):
        print('natural log only accept positive number')
    else:
        #use frexp to get mantissa and exponent
        mantissa, exponent = np.frexp(x)
        #use the previously calculated cheb coefficients
        global cheb_coeff_cut
        #renomalize the mantissa for chebval
        mantissa_renorm = mantissa*4-3
        mantissa_log =
np.polynomial.chebyshev.chebval(mantissa_renorm,cheb_coeff_cut)
        #from our previous calculation
        log2_val = mantissa_log + exponent
        #convert to natural log
        natural_log = log2_val/np.log2(np.exp(1))
        return natural_log
#test the error of mylog2
y_test_val_true = np.linspace(1,10,1001)
x_test_val = np.exp(y_test_val_true)
y_test_val_cheb = mylog2(x_test_val)
err = np.abs(y_test_val_true-y_test_val_cheb)
print('the std of error is: ', np.std(err))
plt.clf()
plt.xscale('log')
plt.plot(x_test_val,err)
plt.savefig('question_3.png')
plt.show()
```

The output is:

the std of error is: 6.024236269321544e-08



we can see that indeed we have a natural log calculator with error less than 10^{-6}