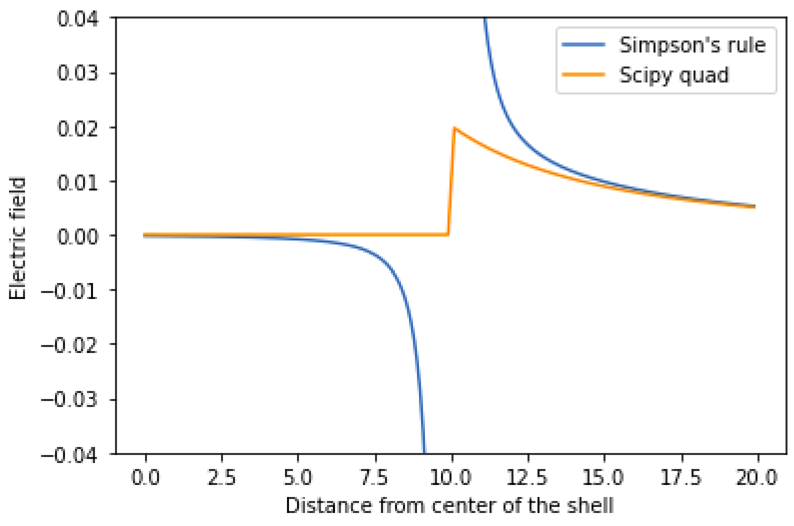
Q1. I take the formula for the electric field of the thin shell of charge density σ and radius R at a distance z from the centre of the shell from Griffiths solutions:

$$\frac{1}{4\pi\epsilon_0}(2\pi R^2\sigma)\int_{-1}^{1}\frac{z-Ru}{(R^2+z^2-2Rzu)^{3/2}}du$$

I take σ such that the factor outside the integral is 1. I take R = 10 units, and z from 0 to 20 units. My code is based on Jon's code which uses the Simpsons rule of integration. I calculate the integration iteratively for different values of z. I also calculate the same integral using scipy quad for the different values of z:

```
R = 10
u = np.linspace(-1, 1, 100)
du = np.median(np.diff(u))
z = np.arange(0, 20, 0.1)
int_quad = np.zeros(len(z))
int_scipy = np.zeros(len(z))
for i in range(len(z)):
    fun = (z[i] - R*u)/(R**2 + z[i]**2 - 2*R*z[i]*u)**1.5
    int_quad[i] = du/3.0*(fun[0]+fun[-1]+4*np.sum(fun[1::2])+2*np.sum(fun[2:-1:2]))
    scipy_fun = lambda x: (z[i] - R*x)/(R**2 + z[i]**2 - 2*R*z[i]*x)**1.5
    int_scipy[i], error = integrate.quad(scipy_fun, -1, 1)
```

The integral and thus the electric field varies like this with the distance from the centre of the shell:



As expected, the E is 0 for z<R and goes like 1/z^2 for z>R. There is a singularity at z=R, which my integration code cares about, but scipy quad ignores.

Q2. My function is based on Jon's function for adaptive step size integration. The idea is that we take 5 equally spaced points with step size dx between the integration limits a and b, and use the Simpson's rule to get a value of integration for it (i2). To estimate the error, we then take a step size of 2dx, and thus use just the 1st, 3rd, and 5th of those equally spaced points (which are now separated by 2dx) to get the integration using Simpson's rule (i1). If i2 and i1 differ by more than the tolerance, we split the interval and two halves and call the function on them recursively with tol=tol/2 until i2-i1<tol.

To avoid calling the function multiple times for the same x, I use the fact that the first call evaluates f(x) at a, a+dx, a+2dx, a+3dx, a+4dx=b. When the function is called recursively on the first half of the interval, dx will be halved and the function values will be needed at a, a+dx/2, a+dx, a+3dx/2, a+2dx. Since, the function is already evaluated at a, a+dx and a+2dx in the previous call, I pass that information while calling the function next. Similarly, I pass a+2dx, a+3dx, a+4dx while calling the function on the second half of the interval:

```python
def integrate_adaptive(fun,a,b,tol,extra=None):
    fun_count = 0
    x=np.linspace(a,b,5)
    dx=x[1]-x[0]
    if extra == None:
        y=fun(x)
        fun_count += len(x)
    else :
        y = [extra[0], fun(x[1]), extra[1], fun(x[3]), extra[2]]
        fun_count += 2
    #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,fun_count
    else:
        mid=(a+b)/2
        int1, count1=integrate_adaptive(fun,a,mid,tol/2, extra=[y[0], y[1], y[2]])
        int2, count2=integrate_adaptive(fun,mid,b,tol/2, extra=[y[2], y[3], y[4]])
        return int1+int2, count1+count2
```

As can be seen, my function has much lesser function call than the lazy approach we did in class:

1) For e^x between 0 and 2:

```python
# Integrating e^x from 0 to 2
from scipy import integrate
ans1, count1 = integrate_adaptive(np.exp, 0, 2, tol = 1e-6, extra=None)
ans2, count2 = integrate_lazy(np.exp, 0, 2, tol = 1e-6)
print("Integral = {} and # of function calls = {} for my function".format(ans1, count1))
print("Integral = {} and # of function calls = {} for the lazy function".format(ans2, count2))
print("Real integral value = {}".format(integrate.quad(np.exp, 0, 2)[0]))
```

```
Integral = 6.3890561284651 and # of function calls = 34 for my function
Integral = 6.3890561284651 and # of function calls = 85 for the lazy function
Real integral value = 6.3890560989306495
```

2) For ln(x) between 1 and 10:

```
ans1, count1 = integrate_adaptive(np.log, 1, 10, tol = 1e-6, extra=None)
ans2, count2 = integrate_lazy(np.log, 1, 10, tol = 1e-6)
print("Integral = {} and # of function calls = {} for my function".format(ans1, count1))
print("Integral = {} and # of function calls = {} for the lazy function".format(ans2, count2))
print("Real integral value = {}".format(integrate.quad(np.log, 1, 10)[0]))
```

```
Integral = 14.025850907644363 and # of function calls = 76 for my function
Integral = 14.025850907644363 and # of function calls = 190 for the lazy function
Real integral value = 14.025850929940457
```

3) For sin(x) between 0 and pi/2

```
ans1, count1 = integrate_adaptive(np.sin, 0, np.pi, tol = 1e-6, extra=None)
ans2, count2 = integrate_lazy(np.sin, 0, np.pi, tol = 1e-6)
print("Integral = {} and # of function calls = {} for my function".format(ans1, count1))
print("Integral = {} and # of function calls = {} for the lazy function".format(ans2, count2))
print("Real integral value = {}".format(integrate.quad(np.sin, 0, np.pi)[0]))
```

```
Integral = 2.0000000217516316 and # of function calls = 48 for my function
Integral = 2.0000000217516316 and # of function calls = 120 for the lazy function
Real integral value = 2.0
```

Q3. I write a function which takes in x from 0.5 to 1, rescales it to get x from -1 to 1. Then, the function starts with fitting a Chebyshev polynomial of order 150 to y = log2(x). Since we want an accuracy better than 10^-6, the function iteratively reduces the order of the Chebyshev polynomial until the maximum error of the fit, that is, the maximum of y_predicted - y_real is just less than 10^-6:

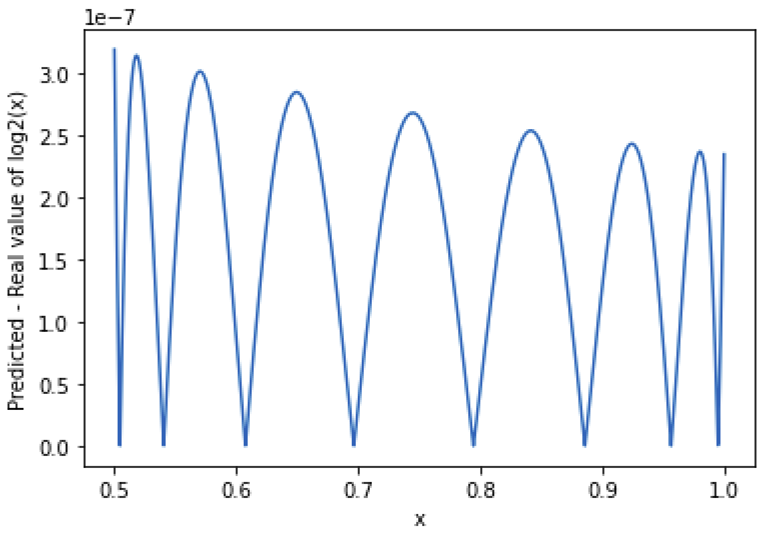
```
def cheby_model(x, y, tol, ord):
    len_x = len(x)
    x_min = x[0]
    x_max = x[len(x)-1]
    old_range = x_max - x_min
    new_range = 1 - (-1)
    new_x = (((x - x_min) * new_range) / old_range) + (-1)
    mat=np.zeros([len_x,ord+1])
    mat[:,0]=1.0
    if ord>0:
        mat[:,1]=new_x
    if ord>1:
        for i in range(1,ord):
            mat[:,i+1]=2*new_x*mat[:,i]-mat[:,i-1]
    coeff = np.polynomial.chebyshev.chebfit(new_x, y, ord)
    pred = np.dot(mat, coeff)
    error = np.abs(y-pred)
    max_error = np.max(error)
    while(max_error <= tol):
        ord = ord - 1
        mat = mat[:, 0:ord+1:1]
        coeff = coeff[0:ord+1:1]
        pred = np.dot(mat, coeff)
        error = np.abs(y-pred)
        max_error = np.max(error)
        rms_error = np.sqrt(np.mean((pred-y)**2))
    return pred, max_error, rms_error, ord+1, coeff
```

As can be seen, the accuracy of the fit is better than 10^-6 when we take 8 terms in the Chebyshev polynomial (that is, a 7 order Chebyshev polynomial):

```
x = np.linspace(0.5, 1, 10000)
y = np.log2(x)
ord = 150
tol = pow(10, -7)
pred, max_error, rms_error, terms, coeff = cheby_model(x, y, tol, ord)
print("The number of terms needed to have a max error = {} and rms error = {} is {}".format(max_error, rms_error, te
```

```
The number of terms needed to have a max error = 3.196978304309539e-07 and rms error = 1.9185635134716262e-07 is 8
```

The residuals of the fit look like:



Then, I define a function to take the ln(x). For that, for each input x, I call np.frexp(x) which splits x as: x = mant*2^expo. log2 of x is then just log2(mant) + expo. Mant is between 0.5 to 1, so the function defined above can be used to get log2(mant). I do the same procedure to get log2(e). The ln(x) is just log2(x)/log2(e).

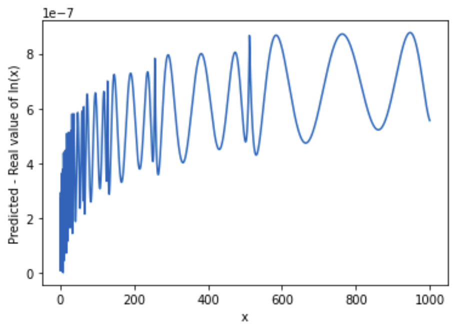
```
def mylog2(x, coeff):
    mant, exp = np.frexp(x)
    mant_new = (mant - 0.5)*(1 -(-1))/(1-0.5) + (-1)
    log2_x = np.polynomial.chebyshev.chebval(mant_new, coeff) + exp
    mant_e, exp_e = np.frexp(np.e)
    mant_e_new = (mant_e - 0.5)*(1 -(-1))/(1-0.5) + (-1)
    log2_e = np.polynomial.chebyshev.chebval(mant_e_new, coeff) + exp_e
    ln = log2_x/log2_e
    return ln
```

The residuals and errors for the ln(x) calculated by the above function looks like:

```
x = np.linspace(0.1, 1000, 10000)
ln_pred = mylog2(x, coeff)
ln = np.log(x)
max_error_ln = np.max(np.abs(ln_pred - ln))
rms_error_ln = np.sqrt(np.mean((ln_pred-ln)**2))
print("Max error = {} and RMS error = {} in the chebyshev fit for ln(x)".format(max_error_ln, rms_error_ln))
plt.plot(x, np.abs(ln_pred - ln))
plt.xlabel("x")
plt.ylabel("Predicted - Real value of ln(x)")
```

```
Max error = 8.768117671920095e-07 and RMS error = 6.324117839567934e-07 in the chebyshev fit for ln(x)
```

```
Text(0, 0.5, 'Predicted - Real value of ln(x)')
```

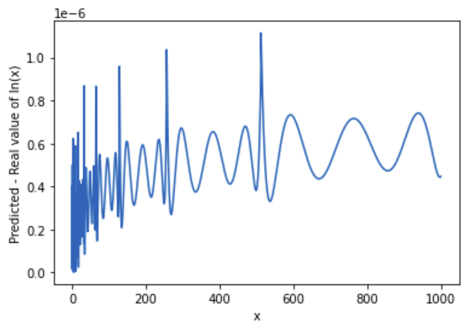


Repeating the same procedure as above for legendre polynomial fit of order 7 (order as returned by my cheby_model function above):

```
x = np.linspace(0.1, 1000, 10000)
ln_pred_leg = mylog2_legendre(x, coeff_legendre)
ln = np.log(x)
max_error_ln_leg = np.max(np.abs(ln_pred_leg - ln))
rms_error_ln_leg = np.sqrt(np.mean((ln_pred_leg-ln)**2))
print("Max error = {} and RMS error = {} in the legendre fit for ln(x)".format(max_error_ln_leg, rms_error_ln_leg))
plt.plot(x, np.abs(ln_pred_leg - ln))
plt.xlabel("x")
plt.ylabel("Predicted - Real value of ln(x)")
```

```
Max error = 1.1141421474292201e-06 and RMS error = 5.426786559419055e-07 in the legendre fit for ln(x)
```

```
Text(0, 0.5, 'Predicted - Real value of ln(x)')
```



As can be seen, the max error is lower for the Chebyshev fit, but the RMS error is lower for the Legendre fit.