Question 1

The electric field generated by an infinitesimally thin spherical shell of charge with radius R at position z is given gy the integral

$$E_z = \frac{2\pi R^2 \sigma}{4\pi\epsilon_0} \int_0^\pi \frac{(z - R\cos(\theta))\sin(\theta)}{(R^2 + z^2 - 2Rz\cos(\theta))^{3/2}} d\theta,$$

which is taken from problem 2.7 of Griffiths' E&M. For simplicity and neatness, let $R = \sigma = \epsilon_0 = 1$, and remove all the constants. We are left with:

$$E_z = \int_0^\pi \frac{(z - \cos(\theta))\sin(\theta)}{(1 + z^2 - 2z\cos(\theta))^{3/2}} d\theta.$$

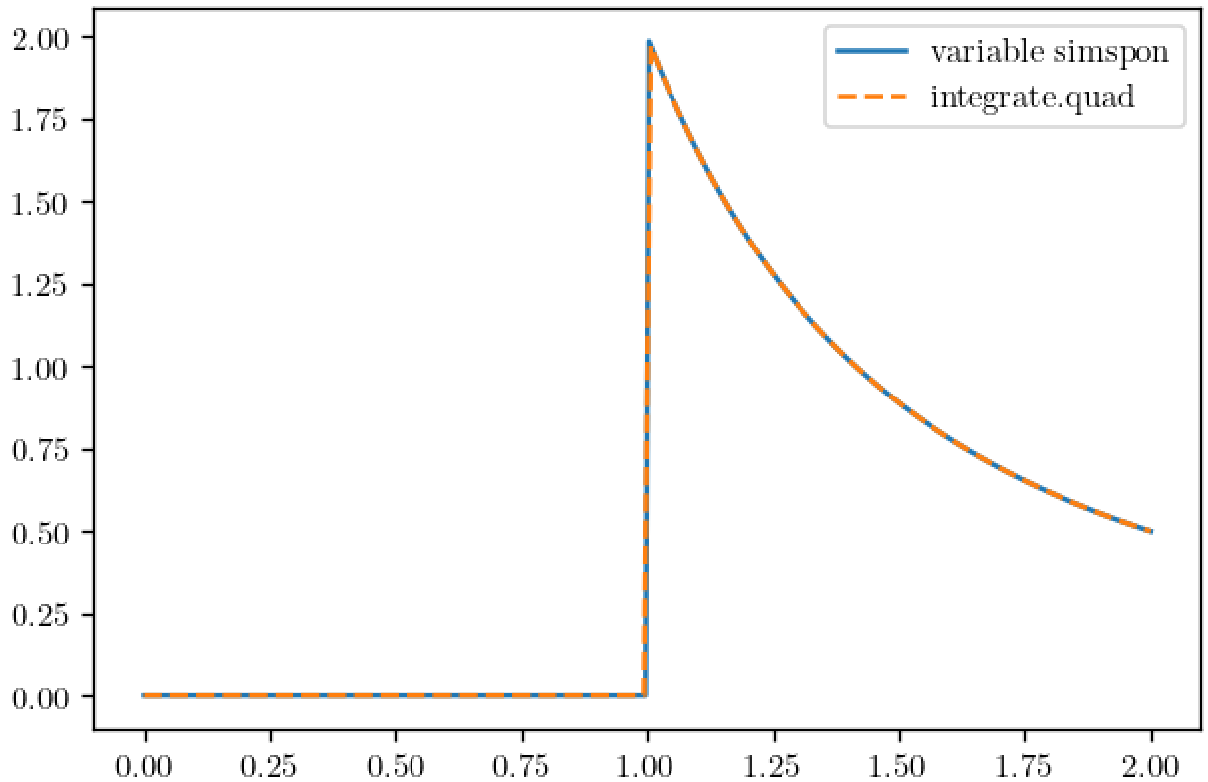
Now let $u = \cos(\theta)$, then $du = -\sin(\theta)$, and Equation (1) is what we need to solve

$$E_z = \int_{-1}^1 \frac{z - u}{(1 + z^2 - 2zu)^{3/2}} du. \tag{1}$$

In [ ]:

```
# using scipy.integrate.quad
import numpy as np
from matplotlib import pyplot as plt
from scipy import integrate

plt.rc('text', usetex=True)
plt.rc('font', family='serif')
plt.rcParams['figure.dpi'] = 120

# defining the function to integrate
def func(u,z):
    return (z - u) / (1 +z**2 -2*z*u)**(3/2)

# bounds of integration
x = [-1,1]
# position in space
z = np.linspace(0, 2, 301)

# getting the values of the integral at all z using scipy.integrate.quad
quad_field = []
for i in z:
    quad_field.append(integrate.quad(func, x[0], x[-1], args = i)[0])
```

In [ ]:
```
# using variable step size simpson from class

def integrate(fun,a,b,tol):
    x=np.linspace(a,b,5)
    dx=x[1]-x[0]
    y=fun(x,i)
    # simpsons rule
    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

# the entire kernel crashes if z = R
# so choose some z1 that does not include z = R
z1 = np.linspace(0, 2, 300)

# call the integrator for all values of z
simpson_field = []
for i in z1:
    def y(u, i):
        return (i - u) / (1 +i**2 -2*i*u)**(3/2)

    simpson_field.append(integrate(y, x[0], x[1], 1e-6))

plt.plot(z1, simpson_field, label = 'variable simspon')
plt.plot(z, quad_field, '--', label = 'integrate.quad')
plt.legend(loc= 'best')
plt.show()
```



There is a singularity in the integral at $z = R$, which is $R = 1$ in this case. On one hand, scipy.integrate.quad does not care at all about the singularity, since it ran perfectly even when I included $z = 1$. On the other hand, the variable step size integrator completely crashes the kernel when it tries to handle $z = 1$. I cannot run anything else until I restart the kernel. To avoid this, I just decided to choose an a range of values of z that did not included 1 (but very close to 1). This worked, and gave pretty much the same result as scipy.intergrate.quad.

Question 2

In [ ]:

```
# Question 2

import numpy as np

# Let's write our integrators

def lazy_integrate(fun,a,b,tol):

    ''' Lazy adaptive Simpsons integrator. Code is tweaked from class.
Takes some function fun, with bounds a and b, and tolerance tol.
    Return the value of the integral, the error and the number of function
calls'''
```

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

    n = len(x)

    # Simpson's rule
    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, myerr, n
    else:
        mid=(a+b)/2
        int_left, myerr_left, n_left = lazy_integrate(fun,a,mid,tol/2)
        int_right, myerr_right, n_right = lazy_integrate(fun,mid,b,tol/2)

        return int_left+int_right, myerr_left+myerr_right, n_right+n_left


def integrate_adaptive(fun, a,b, tol, extra = [None,None,None]):
    ''' Less lazy adaptive Simpsons integrator. Code is again tweaked from
class. Takes some function fun, with bounds a and b, and tolerance tol.
    Return the value of the integral, the error and the number of function
calls'''

    if extra[0] is None:
        x=np.linspace(a,b,5)
        dx=x[1]-x[0]
        y = fun(x)
        n = len(x)
    else:
        x=np.linspace(a,b,5)
        y = fun(x)
        dx=x[1]-x[0]
        y[0], y[2], y[4] = extra
        y[1]= fun(x[1]); y[3]=fun(x[3])
        n = 2
```

```
    # Simpson's rule
    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, myerr, n

    else:
        mid=(a+b)/2
        int_left, myerr_left, n_left = integrate_adaptive(fun,a,mid,tol/2,
[y[0],y[1],y[2]])
        int_right, myerr_right, n_right =
integrate_adaptive(fun,mid,b,tol/2, [y[2],y[3],y[4]])

    return int_left+int_right, myerr_left+myerr_right, n_right+n_left

# defining some functions to test our integrators


def gaussian(x):
    return np.exp(-(x-1)**2/(2))

def sin(x):
    return np.sin(5*x)

def lorentz(x):
    return 1/(1+x**2)
# running the tests



print('Integrating the gaussian from -5 to 5 using the lazy integrator
yields', lazy_integrate(gaussian, -5, 5, tol = 1e-3)[0], 'this has an
error of',
lazy_integrate(gaussian, -5, 5, tol = 1e-3)[1], 'and it
took',lazy_integrate(gaussian, -5, 5, tol = 1e-3)[2], 'functions calls.' )
print('Integrating the gaussian from -5 to 5 using the adaptive integrator
yields', integrate_adaptive(gaussian, -5, 5, tol = 1e-3)[0], 'this has an
```

```
error of',
integrate_adaptive(gaussian, -5, 5, tol = 1e-3)[1], 'and it
took',integrate_adaptive(gaussian, -5, 5, tol = 1e-3)[2], 'functions
calls.' )
print('\n')
print('Integrating the sine from -pi to pi using the lazy integrator
yields', lazy_integrate(sin, -np.pi, np.pi, tol = 1e-3)[0], 'this has an
error of',
lazy_integrate(sin, -np.pi, np.pi, tol = 1e-3)[1], 'and it
took',lazy_integrate(sin, -np.pi, np.pi, tol = 1e-3)[2], 'functions
calls.' )
print('Integrating the sine from -pi to pi using the adaptive integrator
yields', integrate_adaptive(sin, -np.pi, np.pi, tol = 1e-3)[0], 'this has
an error of',
integrate_adaptive(sin, -np.pi, np.pi, tol = 1e-3)[1], 'and it
took',integrate_adaptive(sin, -np.pi, np.pi, tol = 1e-3)[2], 'functions
calls.' )
print('\n')
print('Integrating the lorentzian from -5 to 5 using the lazy integrator
yields', lazy_integrate(lorentz, -5, 5, tol = 1e-3)[0], 'this has an error
of',
lazy_integrate(lorentz, -5, 5, tol = 1e-3)[1], 'and it
took',lazy_integrate(lorentz, -5, 5, tol = 1e-3)[2], 'functions calls.' )
print('Integrating the lorentzian from -5 to 5 using the adaptive
integrator yields', integrate_adaptive(lorentz, -5, 5, tol = 1e-3)[0],
'this has an error of',
integrate_adaptive(lorentz, -5, 5, tol = 1e-3)[1], 'and it
took',integrate_adaptive(sin, 5, 5, tol = 1e-3)[2], 'functions calls.' )
```

```
Integrating the gaussian from -5 to 5 using the lazy integrator yields 2.506546750306
8897 this has an error of 0.0002925344949170274 and it took 75 functions calls.
Integrating the gaussian from -5 to 5 using the adaptive integrator yields 2.50654675
03068897 this has an error of 0.0002925344949170274 and it took 30 functions calls.


Integrating the sine from -pi to pi using the lazy integrator yields -1.4443735078085
258e-16 this has an error of 1.4443735078085258e-16 and it took 5 functions calls.
Integrating the sine from -pi to pi using the adaptive integrator yields -1.444373507
8085258e-16 this has an error of 1.4443735078085258e-16 and it took 5 functions call
s.


Integrating the lorentzian from -5 to 5 using the lazy integrator yields 2.7468293729
004447 this has an error of 0.00033018321968407427 and it took 70 functions calls.
Integrating the lorentzian from -5 to 5 using the adaptive integrator yields 2.746829
3729004447 this has an error of 0.00033018321968407427 and it took 5 functions calls.
```

Question 3

In [ ]:

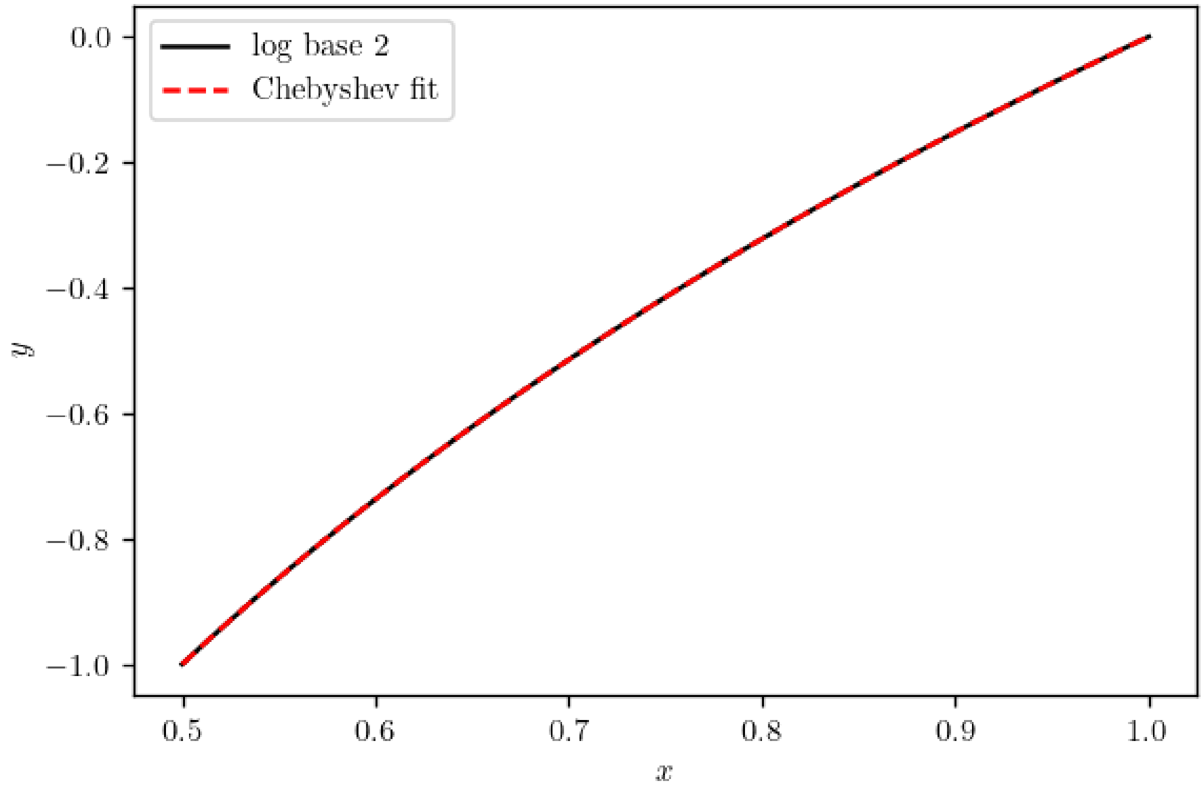
```
# Question 3

import numpy as np
from matplotlib import pyplot as plt
plt.rc('text', usetex=True)
plt.rc('font', family='serif')
plt.rcParams['figure.dpi'] = 120

x = np.linspace(0.5,1,1001)

# rescaling the x-range to [-1, 1] because Chebyshev polynomials are only
defined on [-1, 1]
x_rescaled = 4*x-3

y = np.log2(x)

# getting the chebyshev coefficients up to order 25
cheb_coeffs = np.polynomial.chebyshev.chebfit(x_rescaled,y,25)

print(cheb_coeffs)
```

```
[-4.56893394e-01  4.95054673e-01 -4.24689768e-02  4.85768297e-03
 -6.25084976e-04  8.57981013e-05 -1.22671891e-05  1.80404306e-06
 -2.70834250e-07  4.13047208e-08 -6.37809409e-09  9.94825145e-10
 -1.56462292e-10  2.47798460e-11 -3.94917279e-12  6.32256693e-13
 -1.03000020e-13  1.65751345e-14 -3.48042410e-15  5.78417360e-16
 -1.12041984e-15  1.73786845e-16 -5.57204660e-16  3.08796397e-16
 -5.71175232e-16  1.72741582e-16]
```

The Chebyshev polynomials happen to be bounded by [-1, 1], so the max error we can make by truncating a Chebyshev poly is just the sum of the cut coefficients. We want an error smaller than 1e-6, so we better make sure that the sum of the coefficients we cut off is less than 1e-6. With this in mind, I'll decide to keep up to the 8th term, which is of order 1e-6. All the coefficients that are cut are smaller than 1e-6, so there's no way we have an error greater than 1e-6.

In [ ]:

```
cheb_coeffs_trunc = cheb_coeffs[0:8] # keeping only the 8 first coefficients

# construct the Chebyshev polynomial fit
cheb_poly = np.polynomial.chebyshev.chebval(x_rescaled,cheb_coeffs_trunc)
error = np.abs(cheb_poly-y)
rmse = np.sqrt(np.mean(error**2))

plt.plot(x,y,'k', label = 'log base 2')
plt.plot(x, cheb_poly, 'r--',label = 'Chebyshev fit')
plt.xlabel('$x$')
plt.ylabel('$y$')
plt.legend(loc = 'best')
plt.show()
print('The root mean squared error is: ', rmse, 'and the max error is,', np.max(error))
print('The accuracy is better than 1e-6! So good!')
print(cheb_poly)
```




```
The root mean squared error is:  1.9196336825018886e-07 and the max error is, 3.19697
8212161028e-07
The accuracy is better than 1e-6! So good!
[-9.99999680e-01 -9.98557748e-01 -9.97117254e-01 ... -1.44323762e-03
 -7.21321338e-04  2.35047874e-07]
```

In [ ]:

```python
def mylog2(x):
    ''' returns the natural log of a number x'''
    frexp = np.frexp(x)

    # use our fit to take the log base 2 of the mantissa
    # log base 2 of 2**exponent is just exponent
    log2 = np.log2(frexp[0])+frexp[1]

    # do a change of basis to get natural log
    ln = log2/np.log2(np.exp(1))

    return ln



# examples for some values in [0.5, 1]
print('For x = 0.6, my routine yields', mylog2(0.6), ', the true value
is', np.log(0.6))
```

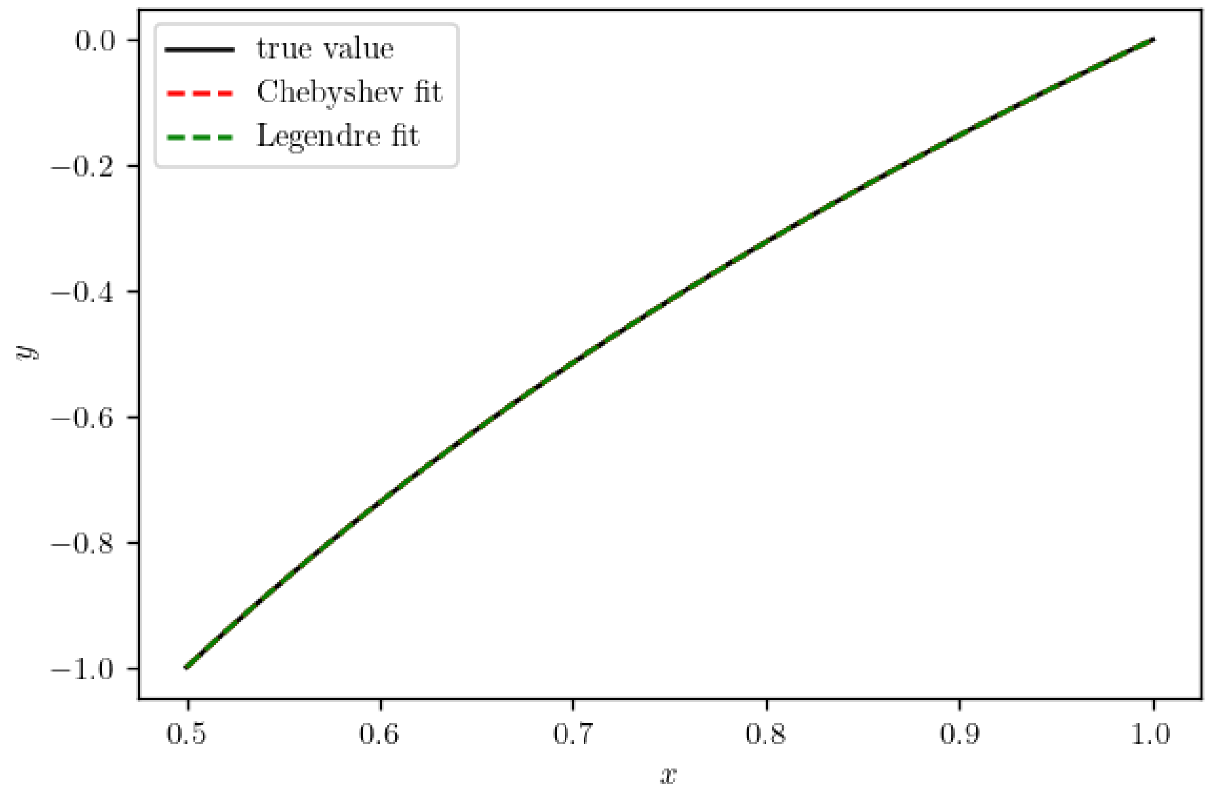
```
print('For x = 0.9, my routine yields', mylog2(0.9), ', the true value
is', np.log(0.9))
```

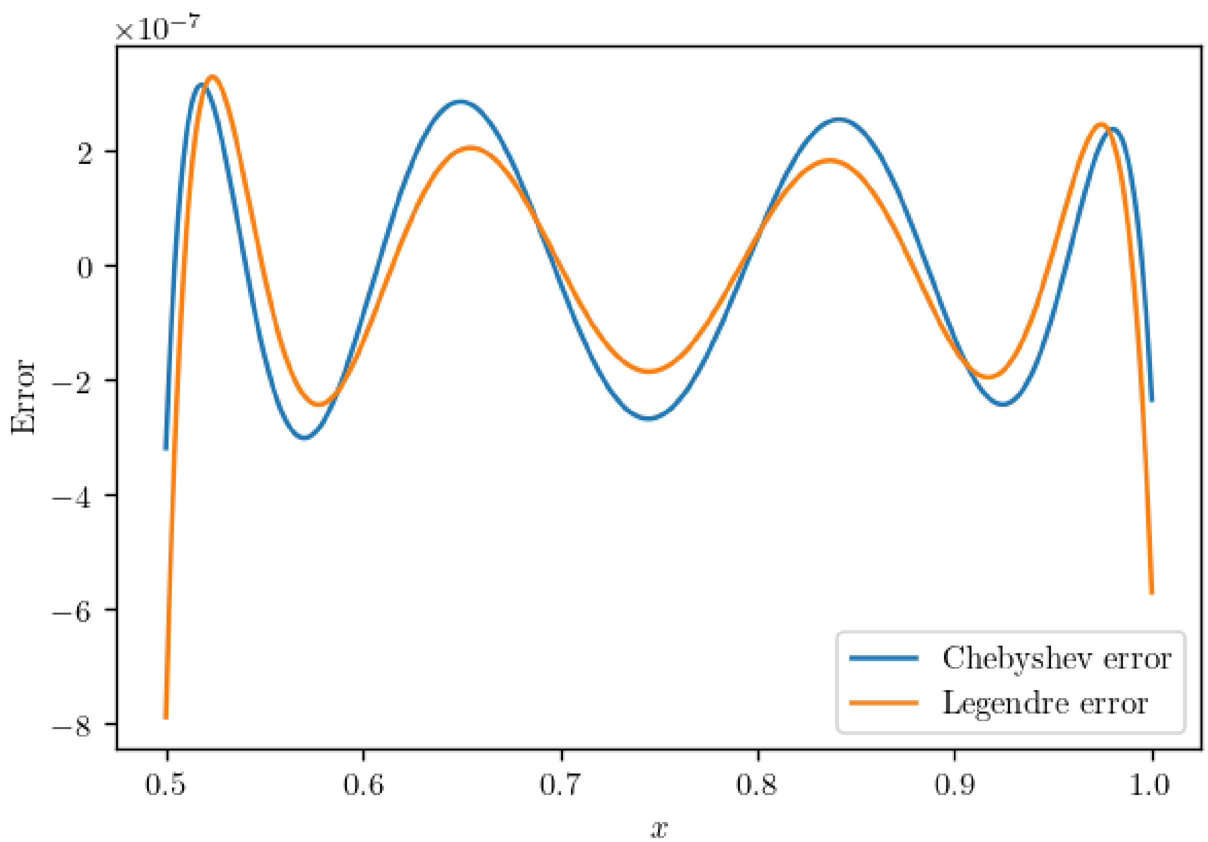
```
For x = 0.6, my routine yields -0.5108256237659907 , the true value is -0.51082562376
59907
For x = 0.9, my routine yields -0.10536051565782628 , the true value is -0.1053605156
5782628
```

It works! Now let's compare our Chebyshev fit to a Legendre fit.

In [ ]:

```
legendre_coeffs = np.polynomial.legendre.legfit(x, y, deg = 7) # Legendre
fit to the same order as Chebshev fit
legendre_poly = np.polynomial.legendre.legval(x,legendre_coeffs)

error_legendre = np.abs(y-legendre_poly)
rmse_legendre = np.sqrt(np.mean(error_legendre**2))

plt.plot(x,y, 'k', label = 'true value')
plt.plot(x, cheb_poly, 'r--', label = 'Chebyshev fit')
plt.plot(x, legendre_poly, 'g--', label = 'Legendre fit')
plt.xlabel('$x$')
plt.ylabel('$y$')
plt.legend(loc='best')
plt.show()

plt.plot(x,y-cheb_poly, label = 'Chebyshev error')
plt.plot(x,y-legendre_poly, label = 'Legendre error')
plt.xlabel('$x$')
plt.ylabel('Error')
plt.legend()
plt.show()

print('Again, for the Chebyshev fit, the root mean squared error is: ',
rmse, 'and the max error is,', np.max(error))
print('For the Legendre fit, the root mean squared error is: ',
rmse_legendre, 'and the max error is,', np.max(error_legendre))
```






```
Again, for the Chebyshev fit, the root mean squared error is:  1.9196336825018886e-07
and the max error is, 3.196978212161028e-07
For the Legendre fit, the root mean squared error is:  1.685237896516845e-07 and the
max error is, 7.888928763577496e-07
```

As we can see from both the numbers just above and from the plot: the Legendre fit as a higher max error than the Chebyshev fit. The root mean squared error are similar for both fits however. Regardless, Chebyshev polynomials are awesome!