PHYS-512, PS2

Muath Hamidi

September, 2022

Problem 1

Well, we can recall the electric field

$$E(r) = \frac{1}{4\pi\epsilon_0}\int \frac{\hat{\mathscr{r}}}{\mathscr{r}^2}dq \tag{1}$$

Substitute the model,

$$E_z = \frac{1}{4\pi\epsilon_0}\int \frac{z - R\cos\theta}{(R^2 + z^2 - 2Rz\cos\theta)^{3/2}}\sigma R^2 \sin(\theta)d\theta d\phi \tag{2}$$

Integrate over ϕ, and let $u = \cos\theta$

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

Now, use the code:

```
#==================================================
# Course: PHYS 512
# Problem: PS2 P1
#==================================================
# By: Muath Hamidi
# Email: muath.hamidi@mail.mcgill.ca
# Department of Physics, McGill University
# September 2022

#==================================================
# Libraries
#==================================================
import numpy as np # For math
import matplotlib.pyplot as plt # For graphs
from scipy.integrate import quad # For math

#==================================================
# Parameters
#==================================================
# Take the whole constant beside the integral --> 1
R = 1 # R
Z = np.linspace(0, 3*R, 401) # z values, from 0 to 3R
u1 = -1 # Integral's lower limit
u2 = 1 # Integral's upper limit

#==================================================
# Integrand
#==================================================
```

```
def Integrand(u):
    Integrand = (z - R*u) / (R**2 + z**2 - 2*R*z*u)**(1.5)
    return Integrand

#==================================================
# Integration
#==================================================
def Integration(Integrand,u1,u2,tol): # This is Jon's function with modification
    us = np.linspace(u1,u2,5) # u partition
    du = us[1] - us[0]
    y = Integrand(us)
    area1 = (y[0]+4*y[1]+2*y[2]+4*y[3]+y[4])*du/3 # 3-point with du
    area2 = (y[0]+4*y[2]+y[4])*(2*du)/3 # 3-point with 2du
    Error = np.abs(area1-area2)
    if Error < tol:
        return area1
    else:
        mid = (u1+u2)/2
        int1 = Integration(Integrand, u1, mid, tol/2)
        int2 = Integration(Integrand, mid, u2, tol/2)
        return int1 + int2

# My Integration
MyE = []
for i in range(0, len(Z)):
    z = Z[i]
    if z == 1: # Here where we have singularity
        MyE.append(np.nan)
    else:
        MyE.append(Integration(Integrand,u1,u2,tol=1e-6))

# Quad Integration
Quad = []
for i in range(0, len(Z)):
    z = Z[i]
    ans = quad(Integrand, -1, 1)
    Quad.append(ans[0])

#==================================================
# Plot
#==================================================
plt.plot(Z, Quad, c='green', label="Quad")
plt.plot(Z, MyE, ls=':', lw=5, label="My E")
plt.title("$E$ vs $z/R$")
plt.xlabel("$z/R$")
plt.ylabel("$E$")
plt.legend()
plt.show
plt.savefig('2.1.pdf', format='pdf', dpi=1200)
```

Both numerical solutions agrees with the analytic solution, where we have zero field inside the spherical shell, at $z = R$ it jumps to a value equivalent to the electric field value from a dot charge in the center of the sphere, and then decay in rate of $1/z^2$.

There is a singularity in the integral at $z = R$. It seems that quad doesn't care about it, while my integrator does. That's why we needed to do something about it.

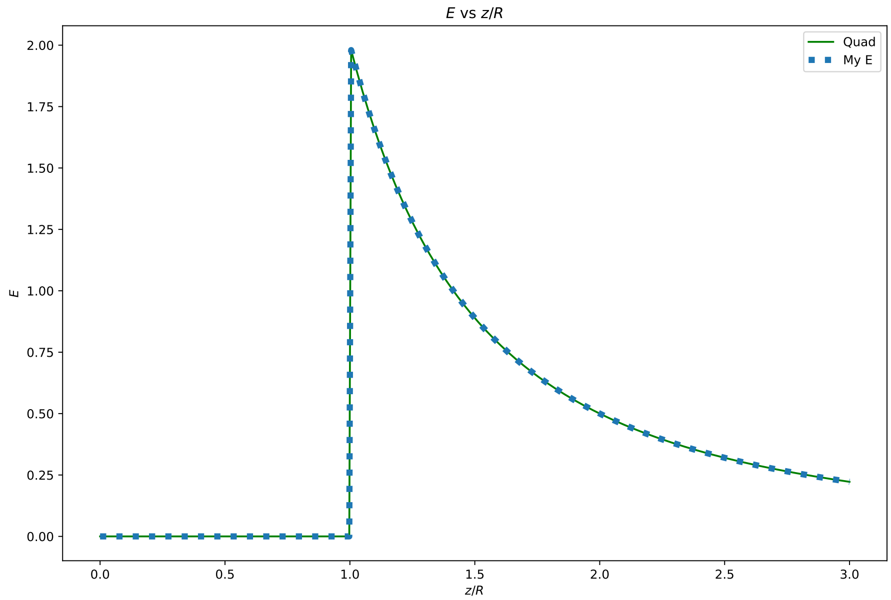

Figure 1: E vs z/R.

Problem 2

Following the Code:

```
#=====================================================
# Course: PHYS 512
# Problem: PS2 P2
#=====================================================
# By: Muath Hamidi
# Email: muath.hamidi@mail.mcgill.ca
# Department of Physics, McGill University
# September 2022

#=====================================================
# Libraries
#=====================================================
import numpy as np # For math
import matplotlib.pyplot as plt # For graphs

#=====================================================
# Functions
#=====================================================
def heaviside(x):
    return 1.0*(x>0)

def offset_gauss(x):
    return 1+10*np.exp(-0.5*x**2/(0.1)**2)

def cos(x):
    return np.cos(x)

```

```python
#=================================================
# Integrate Adaptive
#=================================================
def integrate_adaptive(fun,a,b,tol,extra=None):
    global counter
    if extra==None:
        xs = np.linspace(a,b,5) # x partition
        dx = xs[1] - xs[0]
        y = fun(xs)
        counter += len(y)
        area1 = (y[0]+4*y[1]+2*y[2]+4*y[3]+y[4])*dx/3 # 3-point with dx
        area2 = (y[0]+4*y[2]+y[4])*(2*dx)/3 # 3-point with 2dx
        Error = np.abs(area1-area2)
        if Error < tol:
            return area1
        else:
            mid = (a+b)/2
            int1 = integrate_adaptive(fun, a, mid, tol/2, extra=[y[0], y[1], y[2], dx])
            int2 = integrate_adaptive(fun, mid, b, tol/2, extra=[y[2], y[3], y[4], dx])
            return int1 + int2

    else:
        x = np.array([a+0.5*extra[3],b-0.5*extra[3]])
        y = fun(x)
        counter += len(y)
        dx = extra[3]/2
        area1 = dx*(extra[0]+4*y[0]+2*extra[1]+4*y[1]+extra[2])/3
        area2 = 2*dx*(extra[0]+4*extra[1]+extra[2])/3
        Error = np.abs(area1-area2)
        if Error < tol:
            return area1
        else:
            mid=(a+b)/2
            int1 = integrate_adaptive(fun, a, mid, tol/2, extra = [extra[0] ,y[0], extra[1], dx])
            int2 = integrate_adaptive(fun, mid, b, tol/2, extra = [extra[1] ,y[1], extra[2], dx])
            return int1 + int2

#=================================================
# Integration (Jon's Function)
#=================================================
def Integration(fun,a,b,tol): # This is Jon's function with modification
    global counter
    xs = np.linspace(a,b,5) # x partition
    dx = xs[1] - xs[0]
    y = fun(xs)
    counter += len(y)
    area1 = (y[0]+4*y[1]+2*y[2]+4*y[3]+y[4])*dx/3 # 3-point with dx
    area2 = (y[0]+4*y[2]+y[4])*(2*dx)/3 # 3-point with 2dx
    Error = np.abs(area1-area2)
    if Error < tol:
        return area1
    else:
```

```python
        mid = (a+b)/2
        int1 = Integration(fun, a, mid, tol/2)
        int2 = Integration(fun, mid, b, tol/2)
        return int1 + int2

#===================================================
# Calculations
#===================================================
# Heaviside
# Integration Adaptive
counter = 0
Integ = integrate_adaptive(heaviside, a=0.1, b=1, tol=1e-6, extra=None)
print("Number of function calls, Heaviside, Mine:", counter)
# Jon's Function
counter = 0
Integ = Integration(heaviside, a=0.1, b=1, tol=1e-6)
print("Number of function calls, Heaviside, Jon's:", counter)

# Cos
# Integration Adaptive
counter = 0
Integ = integrate_adaptive(cos, a=-1, b=1, tol=1e-6, extra=None)
print("Number of function calls, Cos, Mine:", counter)
# Jon's Function
counter = 0
Integ = Integration(cos, a=-1, b=1, tol=1e-6)
print("Number of function calls, Cos, Jon's:", counter)

# Gauss
# Integration Adaptive
counter = 0
Integ = integrate_adaptive(offset_gauss, a=-1, b=1, tol=1e-6, extra=None)
print("Number of function calls, Gauss, Mine:", counter)
# Jon's Function
counter = 0
Integ = Integration(offset_gauss, a=-1, b=1, tol=1e-6)
print("Number of function calls, Gauss, Jon's:", counter)
```

The results we have:
Number of function calls, Heaviside, Mine: 5
Number of function calls, Heaviside, Jon's: 5
Number of function calls, Cos, Mine: 65
Number of function calls, Cos, Jon's: 155
Number of function calls, Gauss, Mine: 409
Number of function calls, Gauss, Jon's: 1015

Where in the Heaviside, $x \in [0.1, 1]$. In the Cos, $x \in [-1, 1]$. In the Gauss, $x \in [-1, 1]$.

Problem 3

Our function here is

$$f(x) = \log_2(x) \tag{4}$$

in the interval $x \in [0.5, 1]$. Using np.polynomial.chebyshev.chebfi
t in the code:

```python
#==================================================
# Course: PHYS 512
# Problem: PS2 P3
#==================================================
# By: Muath Hamidi
# Email: muath.hamidi@mail.mcgill.ca
# Department of Physics, McGill University
# September 2022

#==================================================
# Libraries
#==================================================
import numpy as np # For math
import matplotlib.pyplot as plt # For graphs
import numpy.polynomial.chebyshev as chebyshev
from scipy.special import legendre

#==================================================
# Chebyshev
#==================================================
def Chebyshev(z):
    x = np.linspace(0.5,1,101)
    rescaled_x = 4 * x - 3
    y = np.log2(x)
    order = 7 # Using this degree gives 1e-07 to 1e-06 error magnitude. Which is what we want. Known by plotting later.
    return chebyshev.chebval(4 * z - 3, chebyshev.chebfit(rescaled_x, y, order))

#==================================================
# Plot Chebyshev
#==================================================
z = np.linspace(0.5,1,101)
rms = np.sqrt(np.sum((np.log2(z) - Chebyshev(z))**2)/len(z)) # RMS in Chebyshev
print("RMS in Chebyshev:",rms)
plt.plot(z, np.log2(z), label="$log_2$")
plt.plot(z, Chebyshev(z), ls=':', lw=5, label="Chebyshev")
plt.title("Chebyshev")
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.savefig('2.3.1.pdf', format='pdf', dpi=1200)
plt.show()
plt.close()

Error = Chebyshev(z) - np.log2(z)
plt.plot(z, Error) # Error plot
plt.title("Error in Chebyshev")
plt.xlabel("x")
plt.ylabel("y")
plt.savefig('2.3.2.pdf', format='pdf', dpi=1200)
plt.show()
plt.close()

#==================================================
```

```python
# mylog2
#=====================================================
def mylog2(z):
    mantissa, exponent = np.frexp(z)
    x = np.linspace(0.5,1,101)
    rescaled_x = 4 * x - 3
    y = np.log(x)
    order = 7
    cheb = chebyshev.chebval(4 * mantissa - 3, chebyshev.chebfit(rescaled_x, y,
   order))
    return exponent * np.log(2) + cheb

#=====================================================
# Plot mylog2
#=====================================================
z = np.linspace(0.01,100,10000)
rms = np.sqrt(np.sum((np.log(z) - mylog2(z))**2)/len(z)) # RMS in mylog2
print("RMS in mylog2:",rms)
plt.plot(z, np.log(z), label="$log$")
plt.plot(z, mylog2(z), ls=':', lw=5, label="mylog2")
plt.title("mylog2")
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.savefig('2.3.3.pdf', format='pdf', dpi=1200)
plt.show()
plt.close()

Error = mylog2(z) - np.log(z)
plt.plot(z, Error) # Error plot
plt.title("Error in mylog2")
plt.xlabel("x")
plt.ylabel("y")
plt.savefig('2.3.4.pdf', format='pdf', dpi=1200)
plt.show()
plt.close()

#=====================================================
# Legendre Polynomial (Bonus)
#=====================================================
def Legendre(z):
    x = np.linspace(0.5,1,101)
    rescaled_x = 4 * x - 3
    order = 7
    y = legendre(order)(z)
    cheb = chebyshev.chebval(4 * z - 3, chebyshev.chebfit(rescaled_x, y, order))
    return cheb

#=====================================================
# Plot Legendre
#=====================================================
order = 7
z = np.linspace(0.5,1,101)
rms = np.sqrt(np.sum((legendre(order)(z) - Legendre(z))**2)/len(z)) # RMS in
   Legendre
print("RMS in Legendre:",rms)
```

```python
plt.plot(z, legendre(order)(z), label="$Legendre true$")
plt.plot(z, Legendre(z), ls=':', lw=5, label="Legendre")
plt.title("Legendre")
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.savefig('2.3.5.pdf', format='pdf', dpi=1200)
plt.show()
plt.close()

Error = Legendre(z) - legendre(order)(z)
plt.plot(z, Error) # Error plot
plt.title("Error in Legendre")
plt.xlabel("x")
plt.ylabel("y")
plt.savefig('2.3.6.pdf', format='pdf', dpi=1200)
plt.show()
plt.close()
```

The results are:

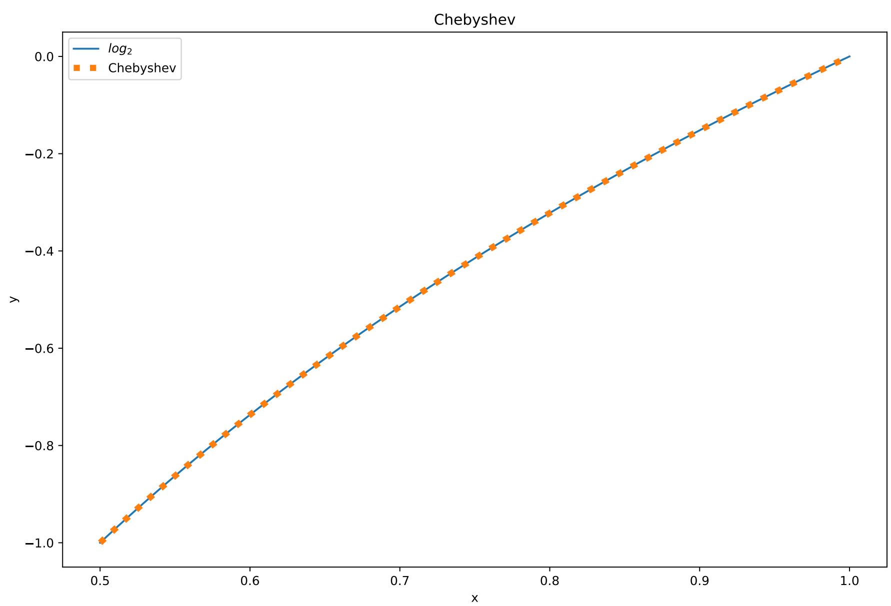

Figure 2: Chebyshev.

Where we have the RMSs:
RMS in Chebyshev: 1.7989069481332663e-07
RMS in mylog2: 1.2150018872283483e-07
RMS in Legendre: 2.2416770311835436e-16

We observe high accuracy in matching the three functions, specially the Legendre polynomial which has accuracy down to machine precision.

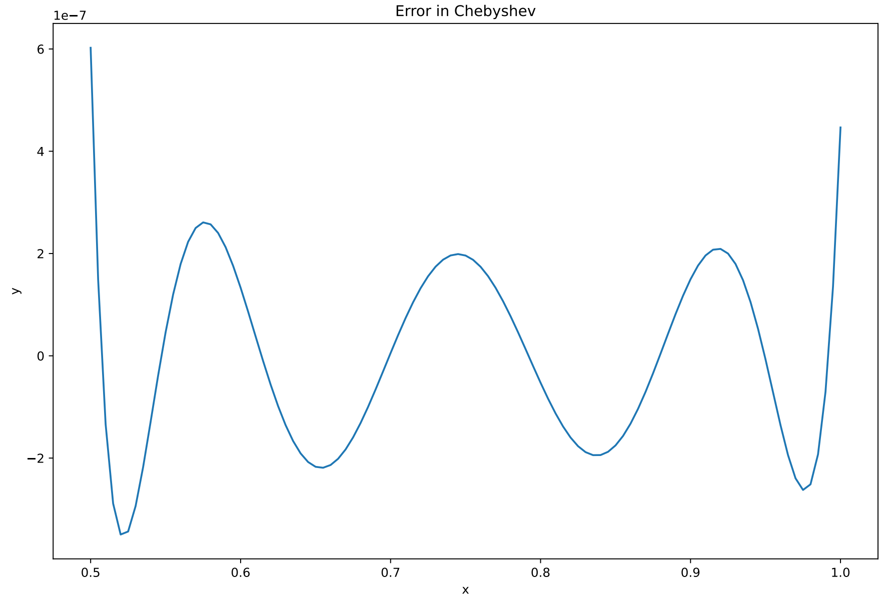

Figure 3: Error in Chebyshev.

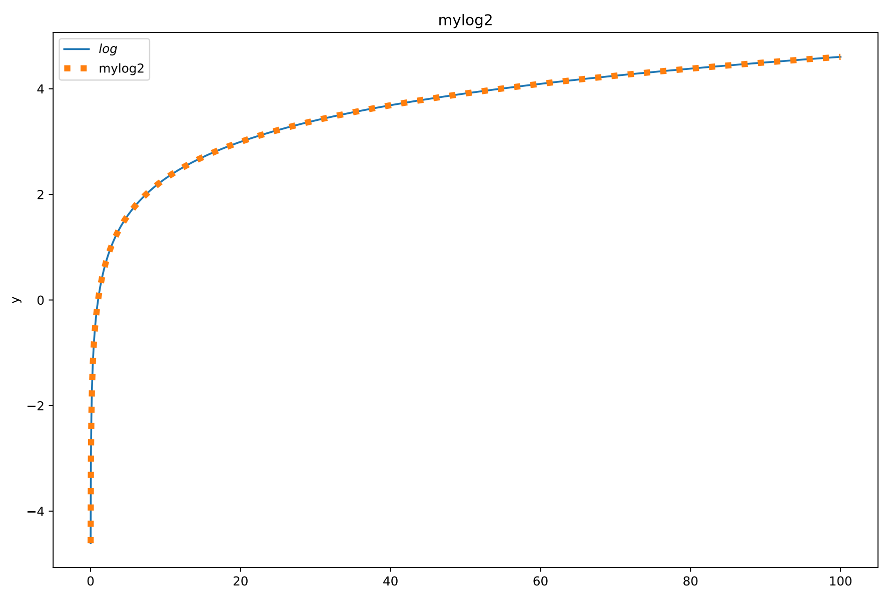

Figure 4: mylog2.

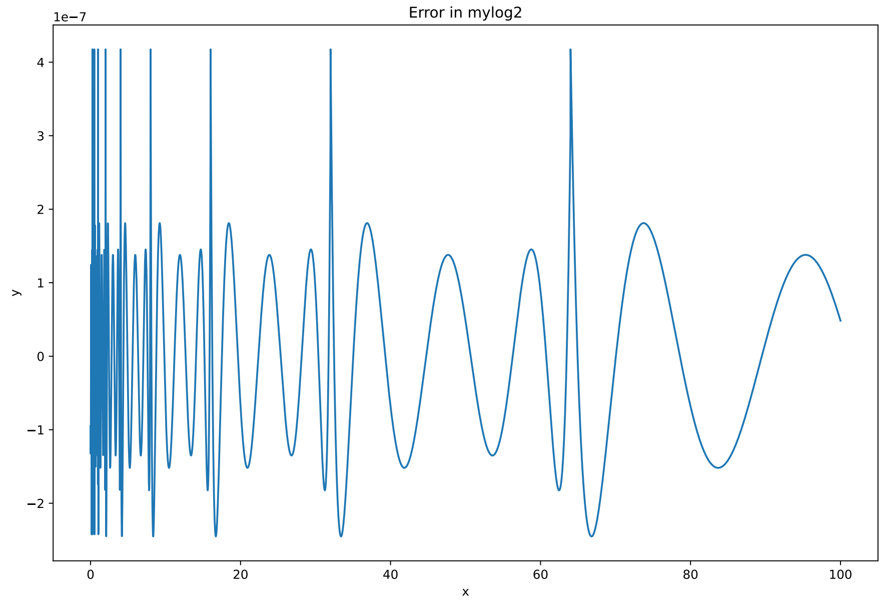

Figure 5: Error in mylog2.

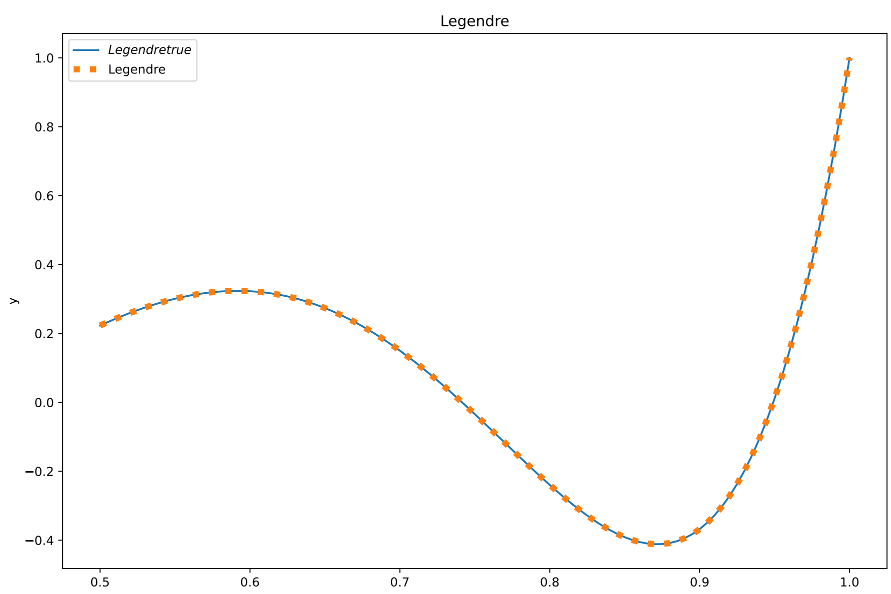

Figure 6: Legendre.

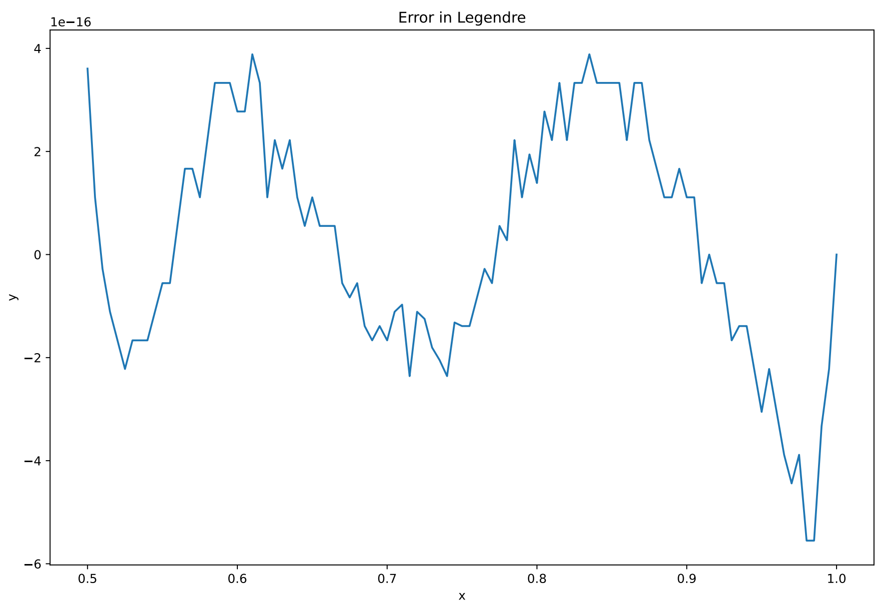

Figure 7: Error in Legendre.