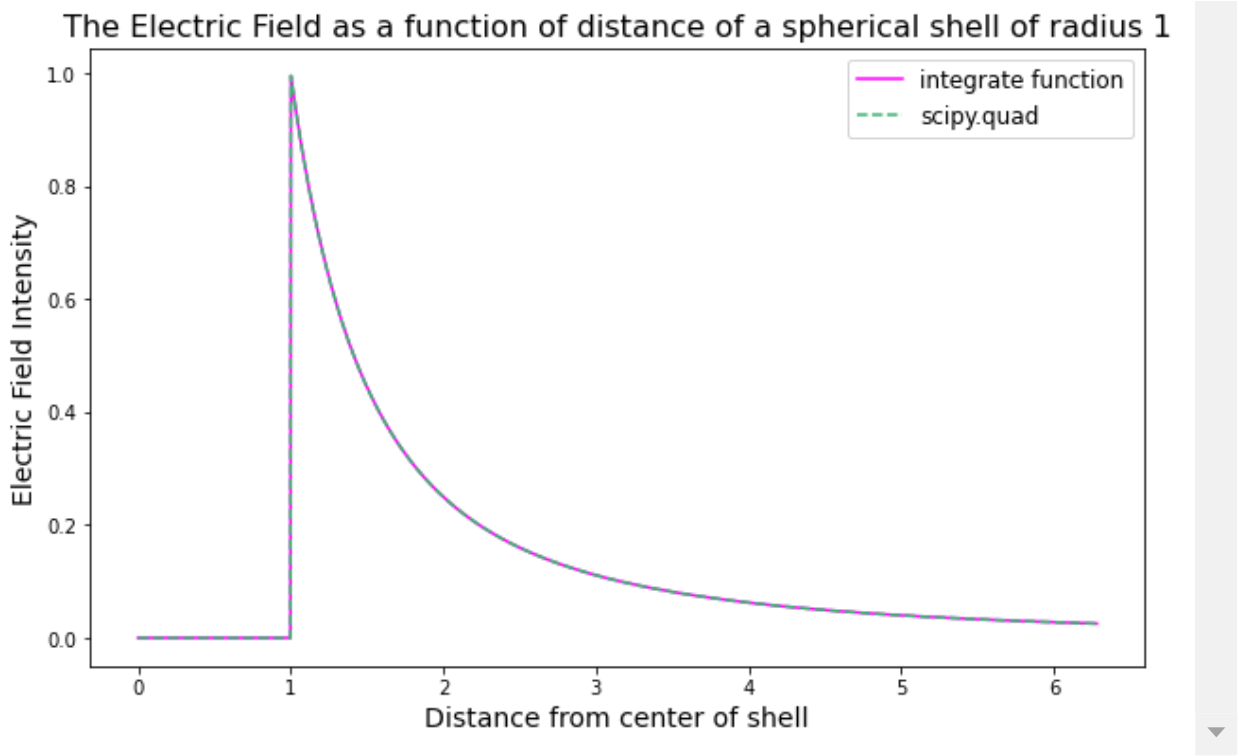
Problem Set 2

Due September 23rd at 11:59pm

Problem 1:

Problem 1: One can work out the electric field from an infinitessimally thin spherical shell of charge with radius R by working out the field from a ring along its central axis, and integrating those rings to form a spherical shell. Use both your integrator and scipy.integrate.quad to plot the electric field from the shell as a function of distance from the center of the sphere. Make sure the range of your plot covers regions with $z < R$ and $z > R$. Make sure one of your z values is R. Is there a singularity in the integral? Does quad care? Does your integrator? Note - if you get stuck setting up the problem, you may be able to find solutions to Griffiths problem 2.7, which sets up the integral.

In [7]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import quad

#Making our integration function
def integrate(fun,a,b,tol):
    #making our 5 points and dx
    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)
    #do the 5-point integral
    i2=(y[0]+4*y[1]+2*y[2]+4*y[3]+y[4])/3*dx
    #Finding the error between the 3 & 5 point integral
    myerr=np.abs(i1-i2)
    #if the error is smaller than our tolerance, then we're set
    if myerr<tol:
        return i2
    #Otherwise we need to split the section and try again
    else:
        mid=(a+b)/2
        int1=integrate(fun,a,mid,tol/2)
        int2=integrate(fun,mid,b,tol/2)
        return int1+int2

#Setting up the integral for the electric field of the spherical shell
def func(theta):
    return (sigma*R**2/(2*eps_0))*(z-R*np.cos(theta))*np.sin(theta)/(R**2+z**2-2*

#defining all the variables in our function
R=1
sigma=1
eps_0=1

x=np.linspace(0,2*np.pi,2001)
y=np.zeros(len(x))
y2=np.zeros(len(x))

#Integrating for multiple values of z
for i in range(len(x)):
    z=x[i]
    y[i]=integrate(func,0,np.pi,1e-6)
    y2[i]=quad(func, 0, np.pi)[0]

plt.figure(figsize=(10,6))
plt.title('The Electric Field as a function of distance of a spherical shell of r
plt.plot(x,y, label='integrate function', color='magenta')
plt.plot(x,y2, label='scipy.quad', linestyle='--', color='mediumseagreen')
plt.xlabel('Distance from center of shell', fontsize=14)
plt.ylabel('Electric Field Intensity', fontsize=14)
plt.legend(fontsize=12)
plt.show()
```



There is a singularity in the integral when z=R, but neither my integrator nor scipy.quad is bothered by this singularity. My assumption is that even though the function itself has a singularity, the integral of said function is still bound, and so both integration methods are still able to properly calculate the integral.

As a sidenote, the plot we obtained makes sense for the electric field of a spherical shell, as inside the shell by gauss' law the electric field is 0, then there is a jump at the z=R and then a dropoff at $\frac{1}{x^2}$ as the shell can be approximated as a point source.

Problem 2:

Problem 2: Write a recursive variable step size integrator like the one we wrote in class that does **NOT** call $f(x)$ multiple times for the same x. The function prototype should be

```
def integrate_adaptive(fun,a,b,tol,extra=None):}
```

where extra contains the information a sub-call needs from preceeding calls. On the initial call, extra should be None, so the integrator knows it is starting off. For a few typical examples, how many function calls do you save vs. the lazy way we wrote it in class?

In [8]:
```
import numpy as np


def integrate_adaptive(fun, a, b, tol, extra=None):
    #determining the x coords of our 5 points and dx
    x=np.linspace(a,b,5)
    dx=x[1]-x[0]

    #Checking if we have an argument for 'extra' to
    #determine whether this is the first time going
    #through the function
    if type(extra)==type(None):
        #Since this is the first time, we need to
        #calculate all y values
        y=fun(x)
        #save the x & y values in an array
        fx=np.array([[x[0],y[0]],[x[1],y[1]],[x[2],y[2]],[x[3],y[3]],[x[4],y[4]]]
    else:
        #if this isn't the first time we go through
        #the function, we first start by making an
        #'empty' array to which we will save our
        #function values
        y=np.zeros(5)
        #Here we go through the x values we want to
        #find f(x) for and check extra to see if we
        #have already calculated f(x) for said x
        for i in range(len(x)):
            for j in range(len(extra[:,0])):
                #If we have already, then no need to
                #calculate again, simply pull the
                #f(x) value from our array
                if x[i]==extra[j,0]:
                    y[i]=extra[j,1]
        #If we haven't then calculate f(x)
        #and add it to our array
        for i in range(len(y)):
            if y[i]==0:
                y[i]=fun(x[i])
                fx=np.vstack((extra,[x[i],y[i]]))

    #Then its like the original integrate function

    #3-point integral
    i3=(y[0]+4*y[2]+y[4])/3*(2*dx)
    #5-point integral
    i5=(y[0]+4*y[1]+2*y[2]+4*y[3]+y[4])/3*dx

    err=np.abs(i5-i3)
    if err<tol:
        return i5
    else:
        mid=(a+b)/2
        int1=integrate_adaptive(fun,a,mid,tol/2,fx)
        int2=integrate_adaptive(fun,mid,b,tol/2,fx)
        return int1+int2
```

In [14]:
```
#Checking how many function calls we save in this new method:

#adding a counter
def integrate_adaptive(fun, a, b, tol, extra=None):
    global adapt_count
    x=np.linspace(a,b,5)
    dx=x[1]-x[0]
    if type(extra)==type(None):
        y=fun(x)
        fx=np.array([[x[0],y[0]],[x[1],y[1]],[x[2],y[2]],[x[3],y[3]],[x[4],y[4]]]
        adapt_count+=5
    else:
        y=np.zeros(5)
        for i in range(len(x)):
            for j in range(len(extra[:,0])):
                if x[i]==extra[j,0]:
                    y[i]=extra[j,1]

        for i in range(len(y)):
            if y[i]==0:
                y[i]=fun(x[i])
                fx=np.vstack((extra,[x[i],y[i]]))
                adapt_count+=1

    i3=(y[0]+4*y[2]+y[4])/3*(2*dx)
    i5=(y[0]+4*y[1]+2*y[2]+4*y[3]+y[4])/3*dx
    err=np.abs(i5-i3)
    if err<tol:
        return i5
    else:
        mid=(a+b)/2
        int1=integrate_adaptive(fun,a,mid,tol/2,fx)
        int2=integrate_adaptive(fun,mid,b,tol/2,fx)
        return int1+int2


#Comparing to our inital function
def integrate(fun,a,b,tol):
    global norm_count
    x=np.linspace(a,b,5)
    dx=x[1]-x[0]
    y=fun(x)
    norm_count+=5
    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


#Trying with a couple examples
```

```
def offset_gauss(x):
    return 1+10*np.exp(-0.5*x**2/(0.1)**2)
def poly(x):
    return 5*x**5+4*x**2+7

norm_count=0
adapt_count=0
integrate(np.cos, -5, 5, 1e-6)
integrate_adaptive(np.cos, -5, 5, 1e-6)
cos_diff=norm_count-adapt_count

norm_count=0
adapt_count=0
integrate(np.exp, 0, 10, 1e-6)
integrate_adaptive(np.exp, 0, 10, 1e-6)
exp_diff=norm_count-adapt_count

norm_count=0
adapt_count=0
integrate(np.sin, 0, 10, 1e-6)
integrate_adaptive(np.sin, 0, 10, 1e-6)
sin_diff=norm_count-adapt_count

norm_count=0
adapt_count=0
integrate(offset_gauss, -5, 5, 1e-6)
integrate_adaptive(offset_gauss, -5, 5, 1e-6)
gauss_diff=norm_count-adapt_count

norm_count=0
adapt_count=0
integrate(poly, -5, 5, 1e-6)
integrate_adaptive(poly, -5, 5, 1e-6)
poly_diff=norm_count-adapt_count

print('Difference in number of function calls for some simple functions:')
print('cos:{}'.format(cos_diff))
print('sin:{}'.format(sin_diff))
print('exp:{}'.format(exp_diff))
print('polynomial:{}'.format(poly_diff))
print('gaussian:{}'.format(gauss_diff))
```

```
Difference in number of function calls for some simple functions:
cos:314
sin:311
exp:1356
polynomial:0
gaussian:422
```

Problem 3:

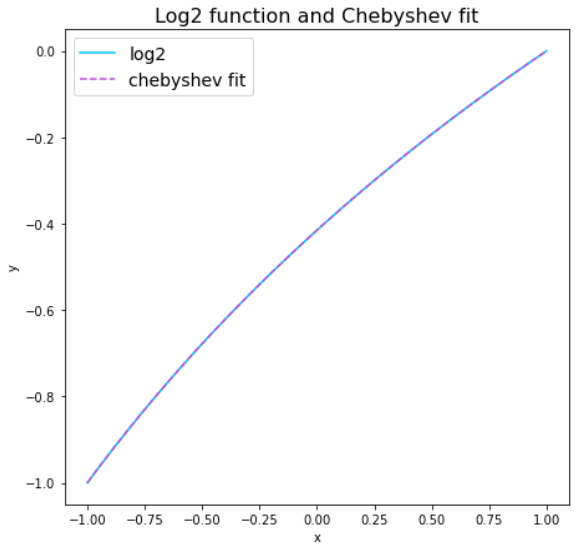
Problem 3: a) Write a function that models the log base 2 of x valid from 0.5 to 1 to an accuracy in the region better than 10^{-6}. Please use a truncated Chebyshev polynomial fit to do this - you can use np.polynomial.chebyshev.chebfit. How many terms do you need? You should use many x/y values and fit to some high order, then only keep the terms you think you'll need and drop the rest. Make sure also that you rescale the x-range you use to go from -1 to 1 before calling chebfit.

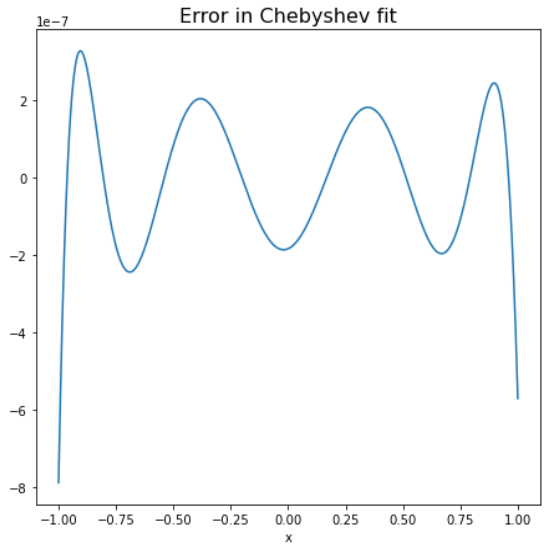
Once you have the Chebyshev expansion for 0.5 to 1, write a routine called mylog2 that will take the natural log of any positive number. Hint - you will want to use the routine np.frexp for this, which breaks up a floating point number into its mantissa and exponent. Also feel free to use np.polynomial.chebyshev.chebval to evaluate your fit. You might ask yourself if a computer ever takes a natural log directly, or if it goes through the log base 2 (near as I can tell, it's the log base 2).

If you'd like a bonus, repeat this exercise using the Legendre polynomial (or, heaven forbid, the actual Taylor series) of the same order as the Chebyshev you used. How big is the RMS error in the log compared to Chebyshev? How big is the maximum error? This hopefully gives you a flavor as to how useful Chebyshev's can be if you have to write your own function. If you enjoy this sort of problem, I'd encourage you to look at http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.24.5177 that describes how some of these transcendental functions are actually implemented.

In [20]:
```
import numpy as np

x=np.linspace(-1,1,1001)
#transposing the range x=[0.5,1] to x=[-1,1]
y=np.log2((x+3)/4)

#after some testing, degree 7 was found to be the smallest degree that still
#gives you an accuracy better than 1e-6
deg=7

#creating the coefficients for the chebyshev fit
coeffs=np.polynomial.chebyshev.chebfit(x,y, deg)

#Creating the chebyshev polynomial for all xs
chebs=np.empty([len(x),deg+1])
chebs[:,0]=1
chebs[:,1]=x
for i in range(1,deg):
    chebs[:,i+1]=2*x*chebs[:,i]-chebs[:,i-1]

#multiplying the chebyshev polynomial matrix by the coefficients
#to make the fit
yt = chebs@coeffs

plt.figure(figsize=(16,7))

plt.subplot(1,2,1)
plt.title('Log2 function and Chebyshev fit', fontsize=16)
plt.xlabel('x')
plt.ylabel('y')
plt.plot(x,y, label='log2', color='deepskyblue')
plt.plot(x,yt, label='chebyshev fit', color='mediumorchid', linestyle='--')
plt.legend(fontsize=14)

plt.subplot(1,2,2)
plt.title('Error in Chebyshev fit', fontsize=16)
plt.plot(x,y-yt)
plt.xlabel('x')

print('The maximum error in the chebyshev fit is {}'.format(max(abs(y-yt))))
```

```
The maximum error in the chebyshev fit is 7.888930774191394e-07
```





In order to perform a truncated chebyshev fit with an accuracy better than 10^{-6}, we needed 7 terms in the chebyshev polynomial.

In [21]:
```
#Function to take ln of any positive number

def mylog2(x, coeffs, deg):
    #using frexp to break up x into mantissa*2**exponent
    num=np.frexp(x)
    #transposing the mantissa into the [-1,1] range that
    #our chebyshev fit is in
    a=num[0]*4-3

    #Creating the chebyshev polynomial with the
    #transposed mantissa
    chebs1=np.empty(deg+1)
    chebs1[0]=1
    chebs1[1]=a
    for i in range(1,deg):
        chebs1[i+1]=2*a*chebs1[i]-chebs1[i-1]
    #Multiplying the chebyshev polynomial by the
    #coefficients we calculated earlier, then add
    #the exponent to x to make it so y=log2(x)
    y = chebs1@coeffs+num[1]

    #doing the same we did for x, but now for e
    #so we can find the natural log
    num2=np.frexp(np.e)
    e=num2[0]*4-3
    chebs2=np.empty(deg+1)
    chebs2[0]=1
    chebs2[1]=e
    for i in range(1,deg):
        chebs2[i+1]=2*e*chebs2[i]-chebs2[i-1]
    e2 = chebs2@coeffs+num2[1]

    #log2(x)/log2(e)=ln(x)
    return y/e2
```

Explaining some of the math from the previous function:

We want to find the natural log of some positive number x using log2. To do so, we can use the property that $ln(x) = \frac{log_2(x)}{log_2(e)}$. So now we need to calculate $log_2(x)$ and $log_2(e)$. We want to use our chebyshev fit to calculate these 2 values, but the range of said fit is only [0.5,1], so we need to decompose x and e such that we only ever take the log2 of numbers in this range. This is where np.frexp comes in handy. np.frexp(x) decomposes x into $mantissa \cdot 2^{exponent}$ and returns an array of [mantissa, exponent]. So $log_2(x)$ becomes $log_2(mantissa \cdot 2^{exponennt}) = log_2(mantissa) + log_2(2^{exponent}) = log_2(mantissa) + exponent$ and since the mantissa is in our range [0.5,1] for positive numbers, we can use our chebyshev fit from earlier to calculate $log_2(mantissa)$!

This means that our function for the natural log becomes $ln(x) = \frac{log_2(mantissa_x)+exponent_x}{log_2(mantissa_e)+exponent_e}$. So we can use our chebyshev fit to calculate $log_2(mantissa_x)$ and $log_2(mantissa_e)$, which allows us to calculate $ln(x)$.