```
In [1]:
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import quad
import astropy.constants as c
from tqdm import tqdm
import time
%matplotlib notebook
```

Problem 1

If we split the charged sphere in little charge elements dq, each dq contributes by $dE = \frac{dq}{4\pi\epsilon_0 r^2}$, where r is the distance between the dq element and the point where we want to compute the field. We can rewrite r by using the law of cosines $r^2 = z^2 + R^2 - 2zRcos(\theta)$. Where z is the distance from the center of the sphere and R the radius of the sphere. We can plug this r^2 in the expression for dE.

$$dE = \frac{dq}{4\pi\epsilon_0(z^2+R^2-2zRcos(\theta))}$$

However, only the electric field in the z direction will survive, the others get cancelled out by symmetry, hence we can write

$dE_z = \frac{dq}{4\pi\epsilon_0(z^2+R^2-2zRcos(\theta))}cos(\beta)$, where β is the angle between the z axis and the segment from dq to the point we evaluate the field at.

and $cos(\beta) = \frac{z-Rcos(\theta)}{r} = \frac{z-Rcos(\theta)}{(z^2+R^2-2zRcos(\theta))^{1/2}}$

Hence $dE_z = \frac{(z-Rcos(\theta))\ dq}{4\pi\epsilon_0(z^2+R^2-2zRcos(\theta))^{3/2}}$

and $dq = \sigma dA = \sigma R^2 sin(\theta)d\theta d\phi$

Combining all of this, $dE_z = \frac{(z-Rcos(\theta))\ \sigma R^2 sin(\theta)d\theta d\phi}{4\pi\epsilon_0(z^2+R^2-2zRcos(\theta))^{3/2}}$

Integrating both sides, we get

$$E(z) = \frac{R^2\sigma}{2\epsilon_0}\int_0^\pi \frac{(z-Rcos(\theta))sin(\theta)d\theta}{(z^2+R^2-2zRcos(\theta))^{3/2}},$$

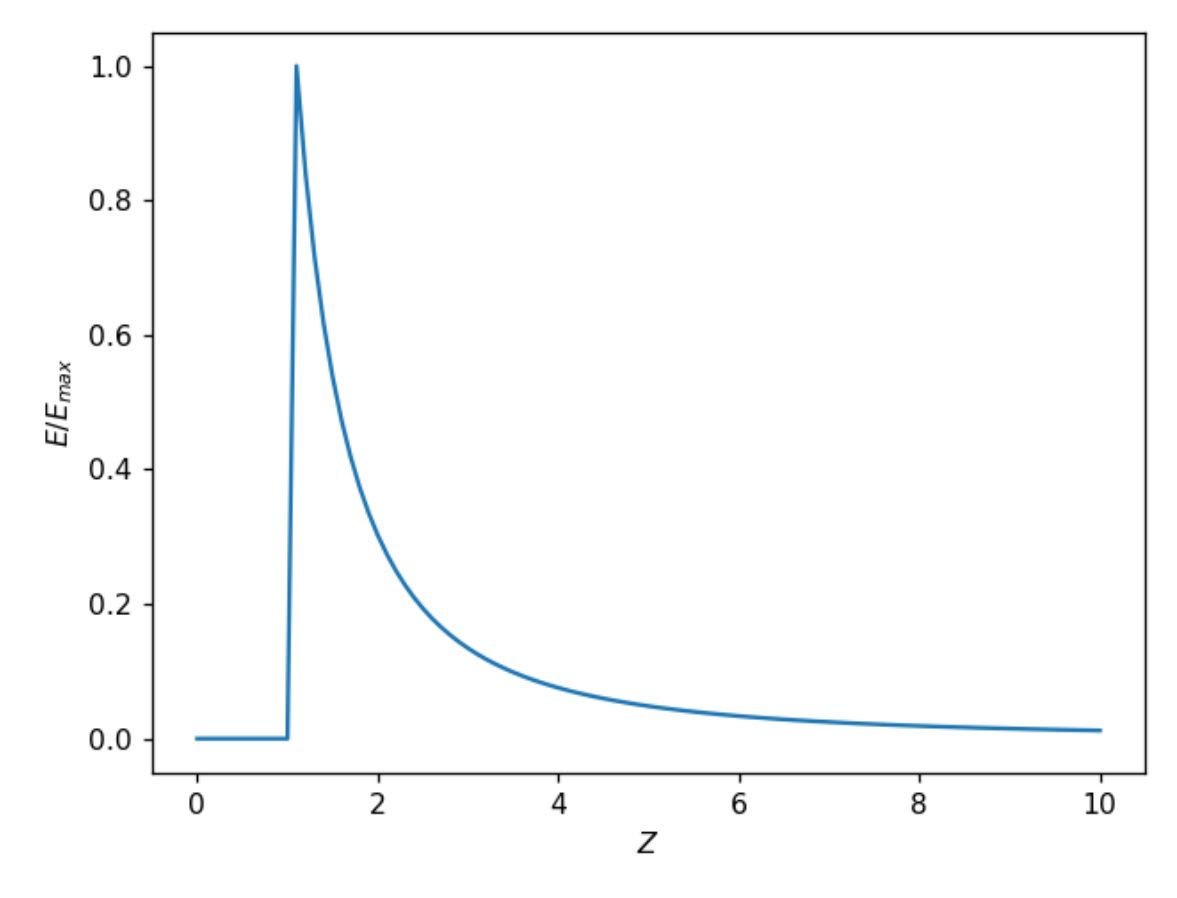
We can solve this integral using numerical methods as below

```
In [82]:
def E(theta,z,R = 1,sigma = 0.01,return_r=False):#defining the function to integrate
    if return_r:
        return R
    return R**2*sigma/(2*c.eps0.value)*((z-R*np.cos(theta))*np.sin(theta))/(R**2+z**2-2*R
*z*np.cos(theta))**(3/2)
```

```
In [83]:
def Ez(fun,z,a,b,tol):#integrator used in the lectures modified such that it evaluates at
 a z value
    x=np.linspace(a,b,5)
    dx=x[1]-x[0]
    y=fun(x,z)
    #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=Ez(fun,z,a,mid,tol/2)
        int2=Ez(fun,z,mid,b,tol/2)
        return int1+int2
```

```
In [84]:
def compute_field(zs=np.linspace(0,10,101)): #computes the electruc field for an array of
 z values
    R = E(1,1,return_r=True)
    E_field = np.zeros(len(zs))
    for i in tqdm(range(len(zs))):
        z=zs[i]
        if z == R: #if we encounter the value of R in the z array, we have a singularity,
 and we can't solve the integral
            print('Skipped evaluation at $z=R$')
            continue #skip this value if this happens
        E_field[i] = Ez(E,z,0,np.pi,1)
    E_field/=np.max(E_field)
    fig, ax = plt.subplots()
    ax.plot(zs,E_field)
    ax.set_xlabel('$Z$')
    ax.set_ylabel('$E/E_{max}$')
    return E_field

E_field = compute_field()
```

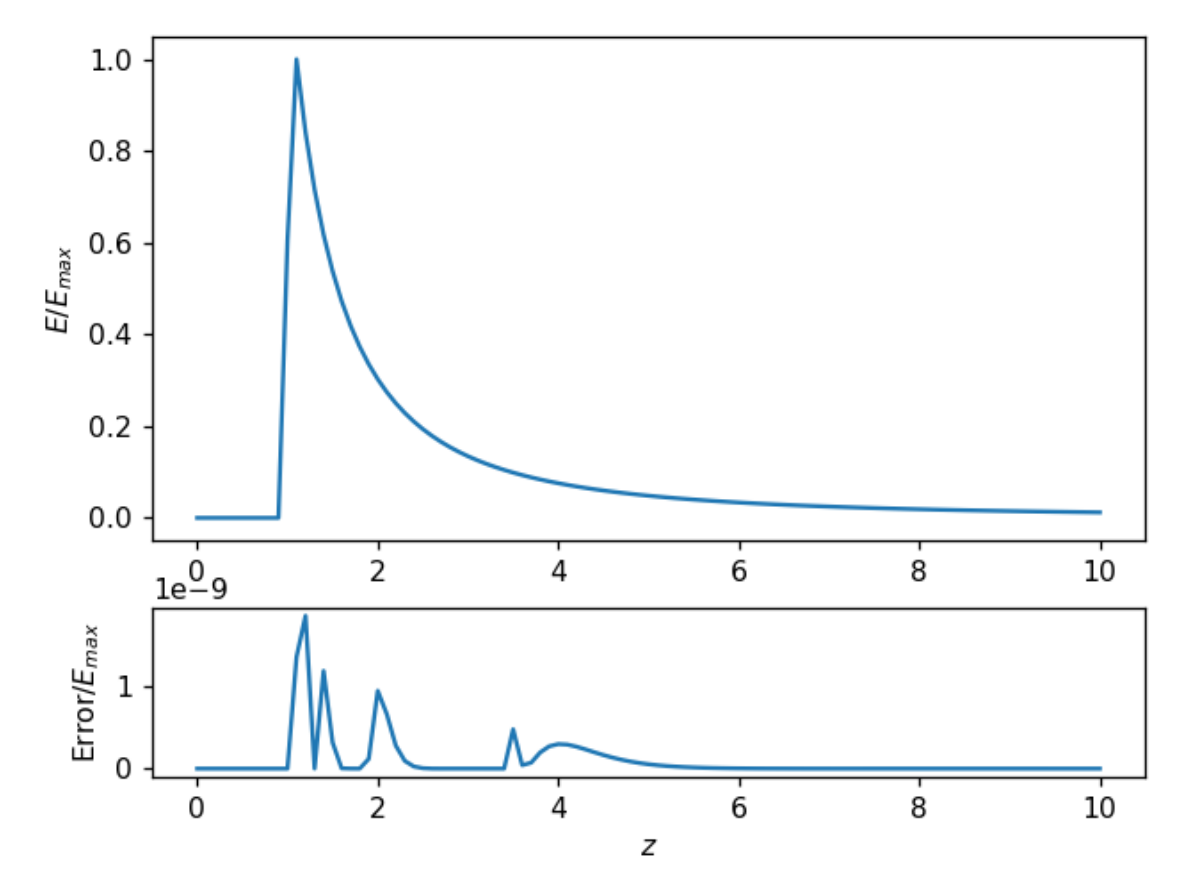
```
 20%|████████████████
| 20/101 [00:00<00:01, 45.50it/s]

Skipped evaluation at $z=R$

100%|████████████████████████████████████████████████████████████████████████████|
101/101 [00:00<00:00, 110.22it/s]
```




```
In [85]:
#now compare with the scipy quad function
zs = np.linspace(0,10,101)

E_field = np.zeros(len(zs))
E_field_err = np.zeros(len(zs))

for i in tqdm(range(len(zs))):
    z=zs[i]

    E_field[i],E_field_err[i] = quad(E,0,np.pi, args = z)

E_field_err/=np.max(E_field)
E_field/=np.max(E_field)
```

```
  0%|
          | 0/101 [00:00<?, ?it/s]<ipython-input-85-6baacf238d6e>:10: IntegrationWarn
ing: The occurrence of roundoff error is detected, which prevents
  the requested tolerance from being achieved.  The error may be
  underestimated.
  E_field[i],E_field_err[i] = quad(E,0,np.pi, args = z)
100%|████████████████████████████████████████████████████████████████████████████|
101/101 [00:00<00:00, 1722.54it/s]
```

```
In [89]:
fig, ax = plt.subplots(2,1, gridspec_kw={'height_ratios': [3, 1]})
ax[0].plot(zs,E_field)
ax[1].set_xlabel('$z$')
ax[0].set_ylabel('$E/E_{max}$')
ax[1].plot(zs,E_field_err)
ax[1].set_ylabel('Error$/E_{max}$')
```




```
Out[89]: Text(0, 0.5, 'Error$/E_{max}$')
```

We can see that both the integrator from the lectures and the scipy.integrate.quad integrator yield the expected results. I.e., the field is 0 inside the shell and goes down as $1/z^2$ when we hit $z = R$. However, there is a singularity at $z = R$ where the electric field is undefined. Quad does not care whether or not there is a z value that is equal to R, but our integrator has trouble dealing with that value because it can never reach the desired tolerance. This is why we have to skip that value for our integrator.

Problem 2

The integrator made in class was making way too many function evaluations. When dividing the interval in 2 parts, we were recalculating all 5 points in the new interval. However, the 5 points form the previous interval are all re-used in the two new smaller intervals (i.e., the first and third point of a large interval bounds the first smaller interval, with the 2nd point of the original interval being the midpoint of the first smaller interval, and similarily for the second small interval). Hence, each time we divide our interval in 2, we only need to do 4 new function evaluations instead of 10. When we want this to happen, we need to have an argument in the function where if we feed it the points from the previous iteration, it uses and only computes the remaining points. If not, we proceed as usual. Each time the extra information is used, we save 3 function calls, hence the counter goes up by three in each block where extra exist. This is what is shown below.

```
In [91]:
def integrate_adaptive(fun,a,b,tol, extra = None):
    x=np.linspace(a,b,5)
    dx=x[1]-x[0]
    if isinstance(extra,tuple): #if extra actually contains information
        ys = extra[0]# the old reusable y values
        counts = extra[1]+3 #counting the saved calls
        y_new = fun(x[1::2]) #evaluating the function only where we dont know it already
        y = np.array([ys[0],y_new[0],ys[1],y_new[1],ys[2]]) #combine the new and old yval
ues
    else: #if it is the first function call, proceed as before
        y=fun(x)
        counts = 0
    #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,counts
    else:
        mid=(a+b)/2
        int1,counts1=integrate_adaptive(fun,a,mid,tol/2,(y[:3],counts)) #pass the first 3
 points to the first smaller interval
        int2,counts2=integrate_adaptive(fun,mid,b,tol/2,(y[2:],counts)) #pass the last 3
 points to the second smaller inteval
        return int1+int2,counts1+counts2
```

We can test the improved integrator below for a couple of functions

```
In [70]:
start = time.time()
ans,saved_calls=integrate_adaptive(np.exp,0,10,1e-8)
stop = time.time()
print('Answer was ',ans, 'with error',ans-(np.exp(10)-np.exp(0)), 'saving', saved_calls,
'calls. Ran in ',stop-start,'seconds')
```

```
Answer was  22025.465794806933 with error 2.1464074961841106e-10 saving 52002 calls.
Ran in  0.13402295112609863 seconds
```

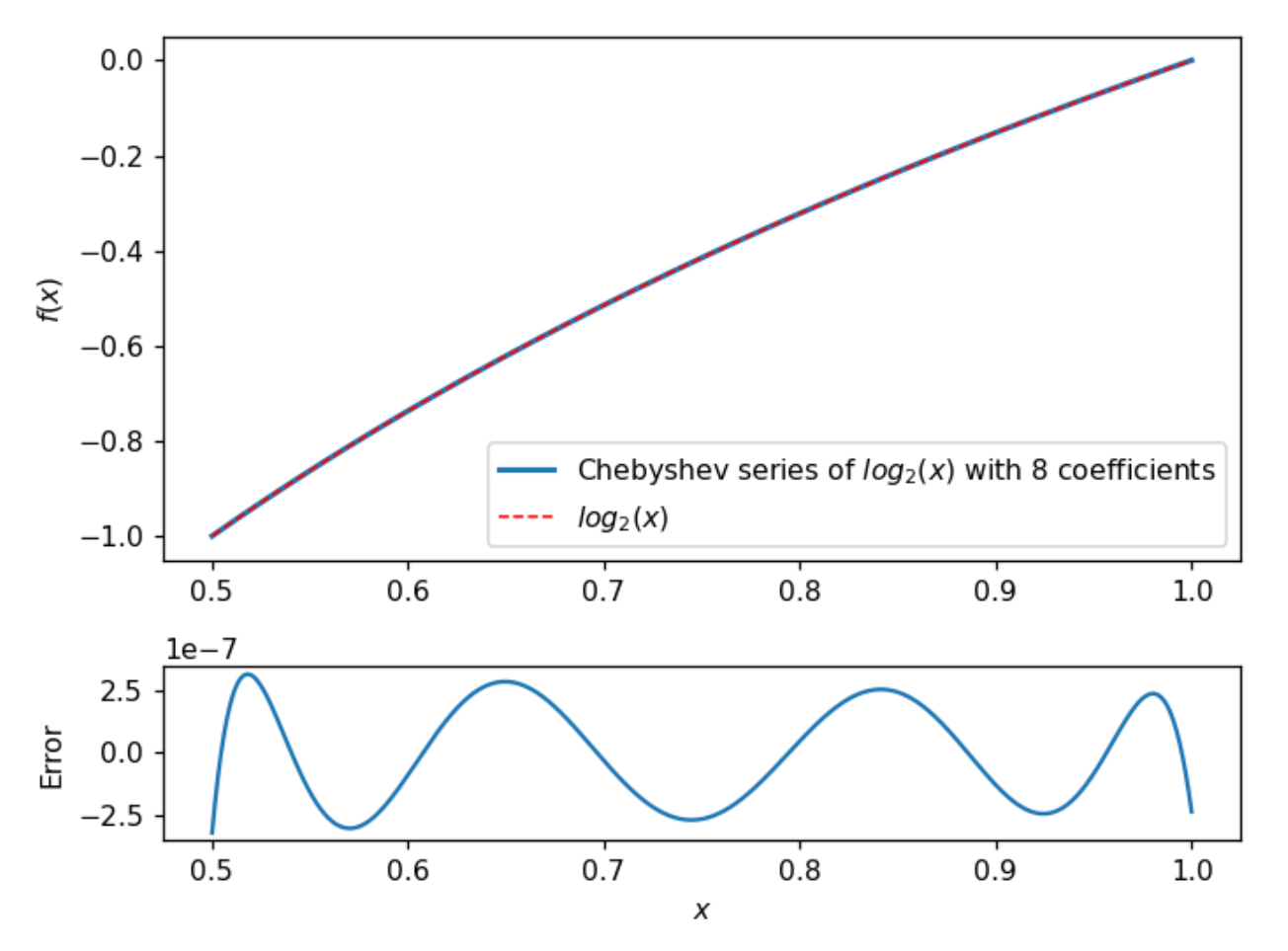
```
In [71]:
start = time.time()
ans,saved_calls=integrate_adaptive(np.sin,0,15,1e-8)
stop = time.time()
print('Answer was ',ans, 'with error',ans-(-np.cos(15)+np.cos(0)), 'saving', saved_calls,
 'calls. Ran in ',stop-start,'seconds')
```

```
Answer was  1.7596879128806284 with error 2.1807000649687325e-11 saving 12987 calls.
Ran in  0.04883980751037598 seconds
```

Problem 3

To model our function with a Chebyshev polynomial, we want $y_i = \sum_j c_j T_j(x_i)$, which can be expressed in matrix form as $y = \tilde{A}x = Ac$, where c is the vector of coefficients, and A is an $m \times n$ matrix where m is the number of points we are fitting to, and n is the order of the polynomial we want to fit. Each column is the j^{th} Chebyshev polynomial evaluated at x_i. i.e, the 3rd column of the matrix has entries corresponding to $T_3(x_i) = 4x_i^3 - 3x_i$. Doing a least squares fit, we can find the coefficients by the following operation $c = (A^TA)^{-1}A^Ty$. We can then evaluate this polynomial for $y = log_2(x)$ between 0.5 and 1. The following function uses np.polynomial.chebyshev.Chebyshev.fit, which returns a Chebyshev object, which is itself a function, so we can evaluate the polynomial directly. It performs the least-squares fit. It also has a trim method that trims the polynomial to the desired accuracy by removing all of the terms with a smaller order of magnitude than the tolerance. We can do this because the error is only as big as the sum of the coefficients. The image of the chebyshev polynomials is bounded between -1 and 1, so the "worst possible error" is if all points happen to evaluate at ± 1 in the polynomials, so the total error is the sum of the coefficients we drop.

```
In [72]:
def model_log2_cheb(a=0.5,b=1,ord = 150,tol=1e-6):
    x = np.linspace(a,b,1001) #create sufficiently large x array
    y = np.log2(x)
    cheb = np.polynomial.chebyshev.Chebyshev.fit(x,y,deg=ord, domain = (a,b)) #perform le
ast squares fit
    cheb = cheb.trim(tol)#remove coeffecients below tolerance
    ncoeffs = len(cheb)
    cheb_eval = cheb(x)#evaluate the polynomial for all x
    xy = np.vstack((x,cheb_eval)).T #the full curve
    return xy, ncoeffs,cheb #return the full x-y points, the number of coefficients and t
he chebyshev object

data,ncoeffs,cheb = model_log2_cheb()
err = np.log2(data[:,0])-data[:,1]
fig,ax = plt.subplots(2,1,gridspec_kw={'height_ratios': [3,1]})
ax[0].plot(data[:,0],data[:,1], linewidth = 2, label = 'Chebyshev series of $log_2(x)$ wi
th {} coefficients'.format(ncoeffs))
ax[0].plot(data[:,0],np.log2(data[:,0]),'--', linewidth = 1, color = 'r', label = '$log_2
(x)$')
ax[1].plot(data[:,0],err)
ax[1].set_xlabel('$x$')
ax[1].set_ylabel('Error')
ax[0].set_ylabel('$f(x)$')
ax[0].legend(loc = 'lower right')
fig.tight_layout()
```

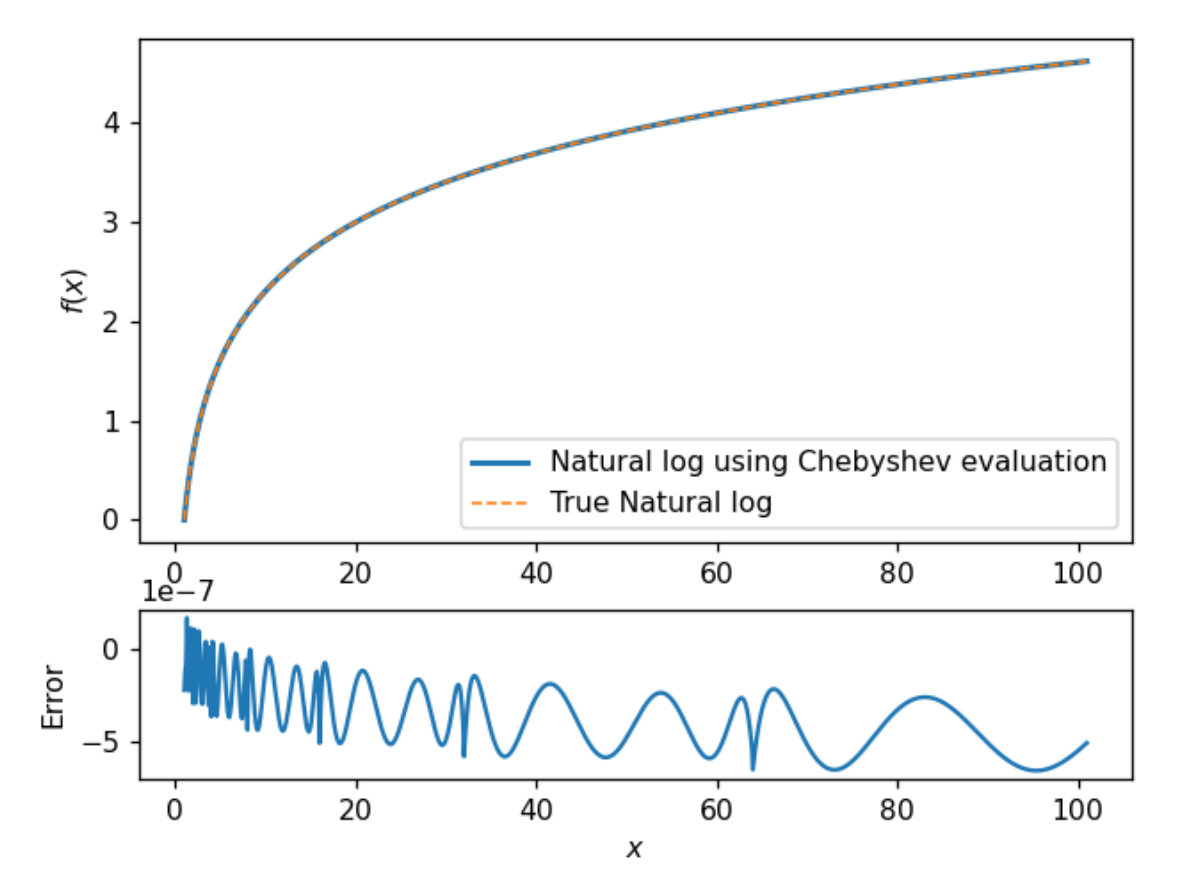


To compute the natural log of any number, we can use our previous function, using the log_2 to get to the ln. When splitting a number into its mantissa and exponent using np.frexp, it expresses it as $a \times 2^b$, where a is between 0.5 and 1. We know that $log_2(a \times 2^b) = log_2(a) + b$, so we can use our previous function to compute $log_2(a)$. We can then change our basis to find the ln: $ln(x) = \frac{log_2(x)}{log_2(e)}$. We need to express e in the same way explained above before taking its log_2, so that our function can handle it.

```
In [78]:
def mylog2(num,tol=1e-6,return_err = False):
    man_num,exp_num = np.frexp(num)#split the number and e in exp-mantissa
    man_e,exp_e = np.frexp(np.e)
    cheb = model_log2_cheb(tol = tol)[2] #get the chebyshev polynomial for the desired to
lerance
    log2_num = cheb(man_num)+exp_num # compute both log_2 as explained above
    log2_e = cheb(man_e)+exp_e
    ln=log2_num/log2_e #change of basis
    err = np.log(num)-ln
    if return_err:
        return ln,err
    else:
        return ln
```

We can test this with an array of values:

```
In [92]:
x = np.linspace(1,101,1001)
lns = mylog2(x,return_err=True)

fig, ax = plt.subplots(2,1,gridspec_kw={'height_ratios': [3,1]})
ax[0].plot(x,lns[0],linewidth = 2, label = 'Natural log using Chebyshev evaluation')
ax[1].plot(x,lns[1])
ax[0].plot(x,np.log(x), linewidth = 1,linestyle='--',label='True Natural log')
ax[0].legend(loc = 'lower right')
ax[1].set_xlabel('$x$')
ax[1].set_ylabel('Error')
ax[0].set_ylabel('$f(x)$')
```




```
Out[92]: Text(0, 0.5, '$f(x)$')
```