Problem Set 1- PHYS512 Computational Physics with Applications

September 20, 2019

Matheus Azevedo Silva Pessôa, ID: 260889844

Problem 1

We saw in class how Taylor series/roundoff errors fight against each other when deciding how big a step size to use when calculating numerical derivatives. If we allow ourselves to evaluate our function f at four points $(x \pm \delta)$ and $x \pm 2\delta)$.

- what should our estimate of the first derivative at x be? Rather than doing a complicated fit, I suggest thinking about how to combine the derivative from $(x \pm \delta)$ with the derivative from $(x \pm 2\delta)$ to cancel the next term in the Taylor series.
- Now that you have your operator for the derivative, what should δ be in terms of the machine precision and various properties of the function? Show for $f(x) = e^x$ and $f(x) = e^{0.01x}$ that your estimate of the optimal δ is at least roughly correct.

First we think of the Taylor expansions for all those points and how to combine them.

$$f(x+\delta) = f(x) + \delta f'(x) + \frac{\delta^2}{2!} f''(x) + \frac{\delta^3}{3!} f'''(x) + \mathcal{O}^4 \tag{1}$$

$$f(x-\delta) = f(x) - \delta f'(x) + \frac{\delta^2}{2!} f''(x) - \frac{\delta^3}{3!} f'''(x) + \mathcal{O}^4 \tag{2}$$

$$f(x+2\delta) = f(x) + 2\delta f'(x) + \frac{(2\delta)^2}{2!} f''(x) + \frac{(2\delta)^3}{3!} f'''(x) + \mathcal{O}^4 \tag{3}$$

$$f(x-2\delta) = f(x) - 2\delta f'(x) + \frac{(2\delta)^2}{2!} f''(x) - \frac{(2\delta)^3}{3!} f'''(x) + \mathcal{O}^4 \tag{4}$$

In order to eliminate third-order terms, we can suvtract,

$$f(x+\delta) - f(x-\delta) = 2\delta f'(x) + \frac{\delta^3}{3!} f'''(x) + ... \tag{5}$$

$$f(x+2\delta) - f(x-2\delta) = 4\delta f'(x) + \frac{8\delta^3}{3!} f'''(x) + ... \tag{6}$$

In order to obtain the first derivative disconsidering third-order terms, we must multiply equation 5 by 8 and them subtract this from the $f(x \pm 2\delta)$ expansion, this obtaining the following result,

$$\boxed{f'(x) = \frac{8f(x+\delta) - 8f(x-\delta) - f(x+2\delta) + f(x-2\delta)}{12\delta}} \tag{7}$$

Writing the code in Python.

```
[10]: # Code for problem 1 A
      #Defining the derivative in function of the first and second order terms in the␣
       ↪Taylor expansion

      def deriv(f,x,h):
          f_prime = (8*f(x+h)-8*f(x-h)-f(x+2*h)+f(x-2*h))/(12*h)
          return f_prime

      #Testing for different values. First, np.exp with epsilon=1E-15.
      deriv(np.exp,np.linspace(0,5),1E-15)
```

```
[10]: array([  1.07321559,   1.11022302,   1.22124533,   1.31376391,
               1.57281595,   1.66533454,   1.90588286,   1.96139401,
               2.18343862,   2.73855013,   3.21964677,   3.44169138,
               3.92278802,   4.36687723,   5.10702591,   5.62512999,
               5.99520433,   6.36527867,   7.40148683,   8.36368012,
               6.66133815,   6.36527867,   8.28966525,   9.02981393,
               9.91799235,  10.65814104,  11.25025998,  13.02661682,
              12.43449788,  16.5793305 ,  15.98721155,  19.83598471,
              21.0202226 ,  22.50051997,  26.05323364,  29.60594732,
              32.56654206,  35.52713679,  39.07985047,  46.18527782,
              53.29070518,  56.84341886,  63.94884622,  71.05427358,
              78.15970093,  93.55479354,  99.47598301, 106.58141036,
             130.26616822, 146.84549872])
```

```
[47]: f = lambda x: x*np.exp(x)
      deriv(f,np.linspace(0,5),1E-15)
```

```
[47]: array([  1.        ,   1.22240181,   1.47104551,   1.77635684,
               2.18343862,   2.49800181,   3.01610588,   3.40468394,
               3.94129174,   5.21804822,   6.40228611,   7.32747196,
               8.80776933,  10.14003696,  11.99040867,  13.98881011,
              15.69115208,  17.31947918,  20.13204418,  24.12884707,
              19.83598471,  21.0202226 ,  26.64535259,  31.67836364,
              34.93501784,  34.93501784,  42.0404452 ,  49.14587256,
              45.59315888,  66.317322  ,  63.94884622,  78.15970093,
              91.18631776,  95.92326933, 117.2395514 , 129.08193033,
             137.37159558, 165.79330501, 180.00415973, 222.63672387,
             270.00623959, 279.48014273, 341.06051316, 364.74527102,
             426.32564146, 516.32772132, 568.43418861, 634.75151061,
             748.43834833, 881.07299234])
```

Here I used 10^{-15} as the error because at 10^{-16} at some point the evaluated values were 0 for some points, and that is due to the machine precision I have on my computer.

Now for the second item, we can consider the total error as the sum of the truncation error (in order of the 5th derivative of the function we used) and the roundoff error. To sum both of them, we can do the following: rewrite the expanded derivatives in terms of small ϵ_i, where $i = (1, 2, 3, 4)$ for the four points we are . Each point has a specific bounded error, and we can rewrite the whole equation considering both types of errors. Mathematically, this can be expressed as:

$$Error = \delta^4 f^{(5)}(x) - \frac{f(x)}{12}(8\epsilon_1 - 8\epsilon_2 - \epsilon_3 + \epsilon_4). \tag{8}$$

The module for the total errors in each derivative plus the truncation error follows the relation $8\epsilon_1 - 8\epsilon_2 + \epsilon_3 - \epsilon_4 < 18\epsilon$, where ϵ represents the complete machine precision we also know. Thus, adding these in module gives us a total of approximately $3\epsilon f(x)/(2\delta)$ for the error we are looking for. Since we are looking to minimize the error and thus find a better suitable δ, what we need to do is derivate the total error in terms of delta and make it equal to zero, thus enabling us to find the minimum value of δ needed to reproduce a good precision (tue *optimal value*).

$$4\delta^3 f^{(5)}(x) - \frac{3\epsilon f(x)}{2\delta^2} = 0 \rightarrow Minimum = \sqrt[5]{\frac{3\epsilon f(x)}{8 f^5(x)}} \tag{9}$$

I was not able to plot the errors in function of delta to find the optimal one, however I do know that it reaches an optimal value before the machine precision one. And that it has the value deduced in this second part in terms of the function itself, machine precsion number and the fifth derivative of $f(x)$.

Problem 2

Lakeshore 670 diodes (successors to the venerable Lakeshore 470) are temperature-sensitive diodes used for a range of cryogenic temperature measurements. They are fed with a constant 10 A current, and the voltage is read out. Lakeshore provides a chart that converts voltage to temperature, available at https://www.lakeshore.com/products/categories/specification/temperature-products/cryogenic-temperature-sensors/dt-670-silicon-diodes, or you can look at the text file I've helpfully copied and pasted (lakeshore.txt). Write a routine that will take an arbitrary voltage and interpolate to return a temperature. You should also make some sort of quantitative (but possibly rough) estimate of the error in your interpolation as well.

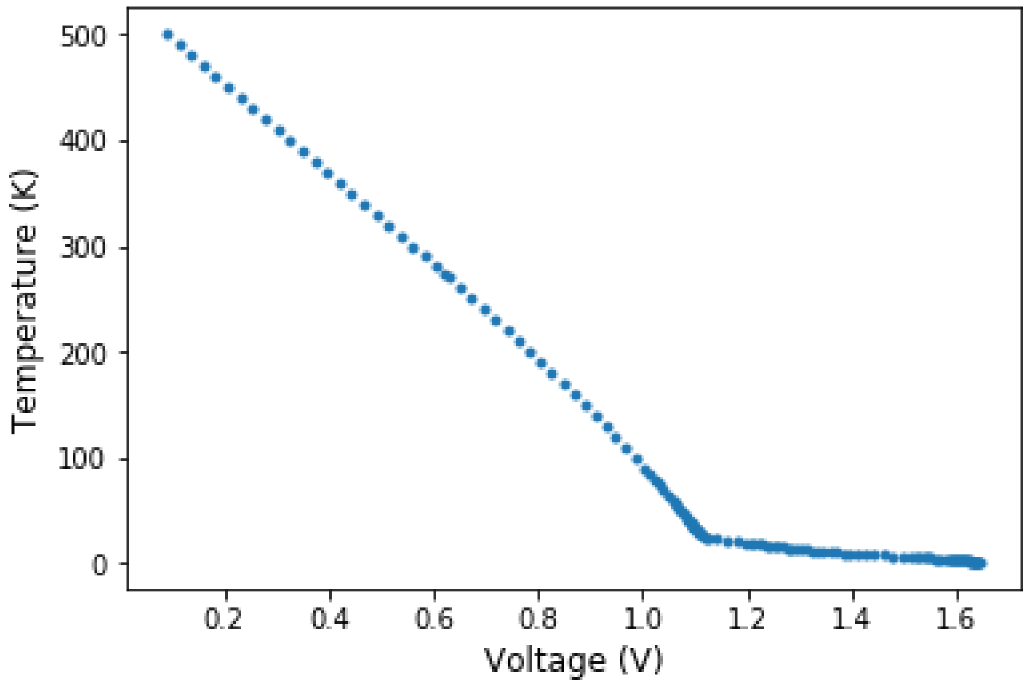
[18]:
```python
#Code for problem 2.
#Some important libraries!
import numpy as np
import matplotlib.pyplot as plt
from scipy import interpolate
import random
```

[19]:

```
#Opening datapoints from lakeshore.txt.
#Taking voltage and temperature data in order to test the interpolation.␣
 ↪However, it is better to work with a list
#so I can sort it and put it into the spline command, since they need to be␣
 ↪sorted out.
#Here I also take a random value in the range of the first and last elements of␣
 ↪voltage because we will use them
#afterwards.
data = np.loadtxt("lakeshore.txt")
temperature = data[:,0]
voltage = data[:,1]
derivative = data[:,2]
voltage=voltage.tolist()
temperature=temperature.tolist()
voltage_sorted=sorted(voltage)
temperature_sorted=sorted(temperature,reverse=True) #To have the same thing again
random_voltage = random.uniform(voltage[0],voltage[-1])
print(random_voltage)
```

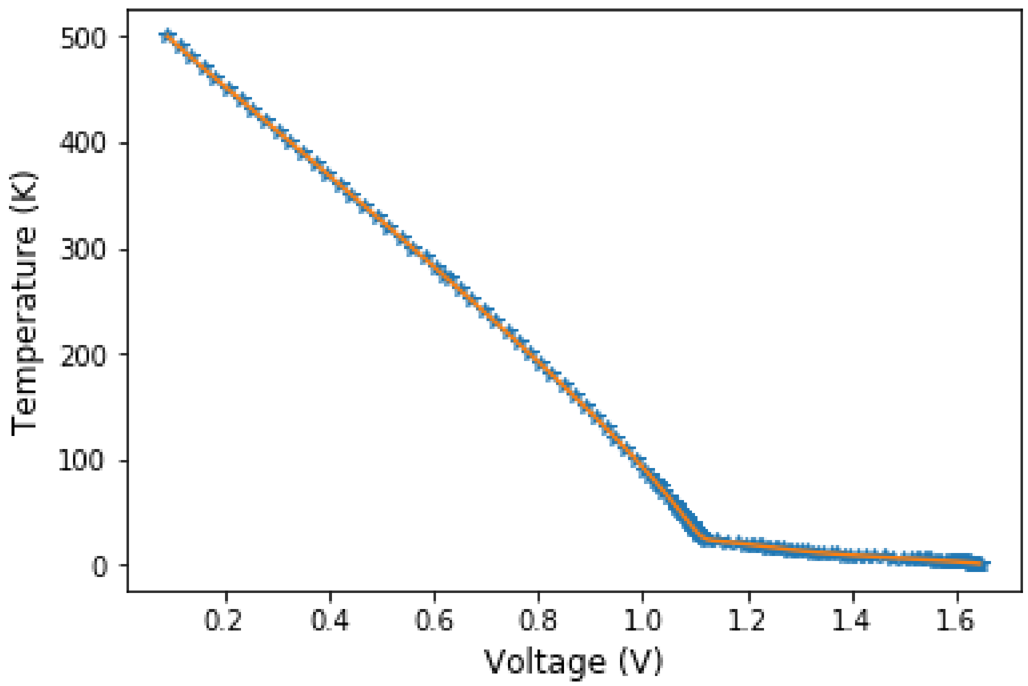
```
0.766748505719
```

[20]:
```
#Plotting the data. This is how temperature depends on voltage.
plt.plot(voltage_sorted,temperature_sorted,".")
plt.xlabel("Voltage (V)",fontsize="large")
plt.ylabel("Temperature (K)",fontsize="large")
plt.show()
```



```
[21]: #Values for variables in the text file. Setting upt to make a spline␣
       ↪interpolation.
      x = voltage_sorted
      y = temperature_sorted
      xx = np.linspace(voltage_sorted[0],voltage_sorted[-1],len(x))
```

```
[22]: #Using a spline interpolation.
      spln = interpolate.splrep(x,y)
      yy = interpolate.splev(xx,spln)
      plt.clf();
      plt.plot(x,y,"*")
      plt.plot(xx,yy)
      plt.xlabel("Voltage (V)",fontsize="large")
      plt.ylabel("Temperature (K)",fontsize="large")
      plt.show()
```



```
[23]: print ("The temperature associated with",random_voltage, " volts is")
      print(interpolate.splev(random_voltage,spln))
```

```
('The temperature associated with', 0.7667485057185466, ' volts is')
207.82350060357737
```

[24]:
```
#Estimating the error roughly. About 31-36% is the value I have found␣
↪considering the comparisons between both
# values for the derivative of temperature in function of voltage and the␣
↪derivative of the spline itself.
#It is indeed a rough estimate.
firstelementderiv=list(derivative)[1]
firstelementinterpolation=list(np.diff(yy))[1]
print(firstelementinterpolation/firstelementderiv)
lastelementinterpolation=list(np.diff(yy))[-1]
lastelementderiv=list(derivative)[-1]
print(lastelementinterpolation/lastelementderiv)
```

```
0.360128418617478
0.31410854326830007
```

These are big errors since we do not know exactly how the derivative values were obtained originally. Another mathematical method or a comparison with another interpolation should be more accurate than using the derivative.

Problem 3

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. For a few typical examples, how many function calls do you save vs. the lazy way we wrote it in class?

[30]:
```
import numpy as np
# The funcion I want to integrate. In this case, exp(x).
def yourfunction(x):
    return np.exp(x)
#This is the integrator we used in class.
def class_integrator(func,a,b,tol):
    x = np.linspace(a,b,5)
    y = func(x)
    # Simpson rule for points 0,2,4 (left,middle,right)
    f1 = (((y[0]+4*y[2]+y[4]))*(b-a))/6.0
    # Simpson rule for each point
    f2 = ((y[0]+4*y[1]+2*y[2]+4*y[3]+y[4])*(b-a))/12.0
    # The error according to Simpson rule
    error = np.abs(f2-f1)/15.0
    neval= len(x)
    if error<tol:
        return (16.0*f2-f1)/15.0,error, neval
    else:
        middle_point = (a+b)/2.0
        left_int,error_left,neval_left =␣
↪class_integrator(func,a,middle_point,tol/2.0)
```

```
        right_int,error_right,neval_right =␣
↪class_integrator(func,middle_point,b,tol/2.0)
        integral = left_int+right_int
        total_error = error_left+error_right
        total_neval = neval+neval_left+neval_right
        return integral, total_error, total_neval

#The new integrator takes into account the points we have already calculated␣
↪instead of recalculating them again
#For this, we call two other
def new_integrator(func,a,b,tol,newx=[],newy=[]):
    # The five points with which we begin
    x = np.linspace(a,b,5)
    y = []
    neval = 0
    #For the first-ever evaluation, there will be no values sitting on the list␣
↪newx[]. Then we can calculate the
    #integral and add the results to this list.
    if len(newx)==0:
        neval+=5
        for i in range(len(x)):
            newx.append(x[i])
            newy.append(func(x[i]))
            y.append(func(x[i]))
    if len(newx)>0:
    # If this is not the first evaluation, we would like to know wheter we do or␣
↪not have a value for a given point
    #i already sitting on the list.
        for i in range(len(x)):
            # If this is true, then we add that to the newx[] list.
            if x[i] in newx:
                y.append(func(x[i]))
            # when this is not true, we will definitely need to call the␣
↪integrator again.
            else:
                newy.append(func(x[i]))
                newx.append(x[i])
                y.append(func(x[i]))
                neval+=1
    # Simpson rule for points 0,2,4 (left,middle,right)
    f1 = (((y[0]+4*y[2]+y[4]))*(b-a))/6.0
    # Simpson rule for each point
    f2 = ((y[0]+4*y[1]+2*y[2]+4*y[3]+y[4])*(b-a))/12.0
    # The error according to Simpson rule
    error = np.abs(f2-f1)/15.0
    if error<tol:
        return (16.0*f2-f1)/15.0,error, neval
    else:
        # if the error is large, we want to use the function again
        # but using different points
        middle_point = (a+b)/2.0
        left_int,error_left,neval_left = new_integrator(func,a,middle_point,tol/
↪2.0,newx,newy)
        right_int,error_right,neval_right =␣
↪new_integrator(func,middle_point,b,tol/2.0,newx,newy)
        integral = left_int+right_int
        total_error = error_left+error_right
        total_neval = neval+neval_left+neval_right
        return integral, total_error, total_neval

integral, error, neval = class_integrator(yourfunction,1,10,1e-10)
print("Integral=",integral,"with",neval,"evaluations, and corresponding␣
↪error",error)
integral2, error2, neval2 = new_integrator(yourfunction,1,10,1e-10)
print("Integral",integral2, "with",neval2,"evaluations, and corresponding␣
↪error", error2)
print ((neval2/neval), "percent advantage")
```

```
('Integral=', 22023.747512978258, 'with', 22535, 'evaluations, and corresponding
error', 3.6688328301319003e-11)
('Integral', 22023.747512978258, 'with', 9017, 'evaluations, and corresponding
error', 3.6688328301319003e-11)
(0, 'percent advantage')
```

Problem 4

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.

To set up the integral, we can start considering the a segment of charge dq and the resulting electric field produced by it as being,

$$\boldsymbol{dE} = \frac{dq}{4\pi\epsilon_0}\frac{1}{\boldsymbol{r}^2}. \tag{10}$$

When considering the case of a spherical surface though, it is important to rewrite the integral all over the surface and calculate the resulting electric field in the z direction, since it is a result of all charge particles around a simple spherical charged shell making the contribution.

One can write the charge density as $dq = \sigma dA$, which in spherical coordinates becomes, $dq = \sigma R^2 \sin\theta d\theta d\phi$. Since $\boldsymbol{E} = E_z$, we have the projection over an auxiliar angle ψ, obtaining $E = E_z \cos\psi = \frac{z - R\cos\theta}{r}$. Now using cosine's law, we get $\boldsymbol{r^2} = R^2 + z^2 - 2Rz\cos\theta$. Integrating all over the surface, we have,

$$E_z = \frac{1}{4\pi\epsilon_0} \int \frac{\sigma R^2 \sin\theta d\theta d\phi (z - R\cos\theta)}{(R^2 + z^2 - 2Rz\cos\theta)^{\frac{3}{2}}} \tag{11}$$

And we use a trigonometric substitution, $u = \cos\theta$, $du = -\sin\theta$ and adjust the integration limits from -1 to 1.

$$E_z = \frac{2\pi R^2 \sigma}{4\pi\epsilon_0} \int_{-1}^{1} \frac{z - Ru}{(R^2 + z^2 - 2Rzu)^{\frac{3}{2}}} du. \tag{12}$$

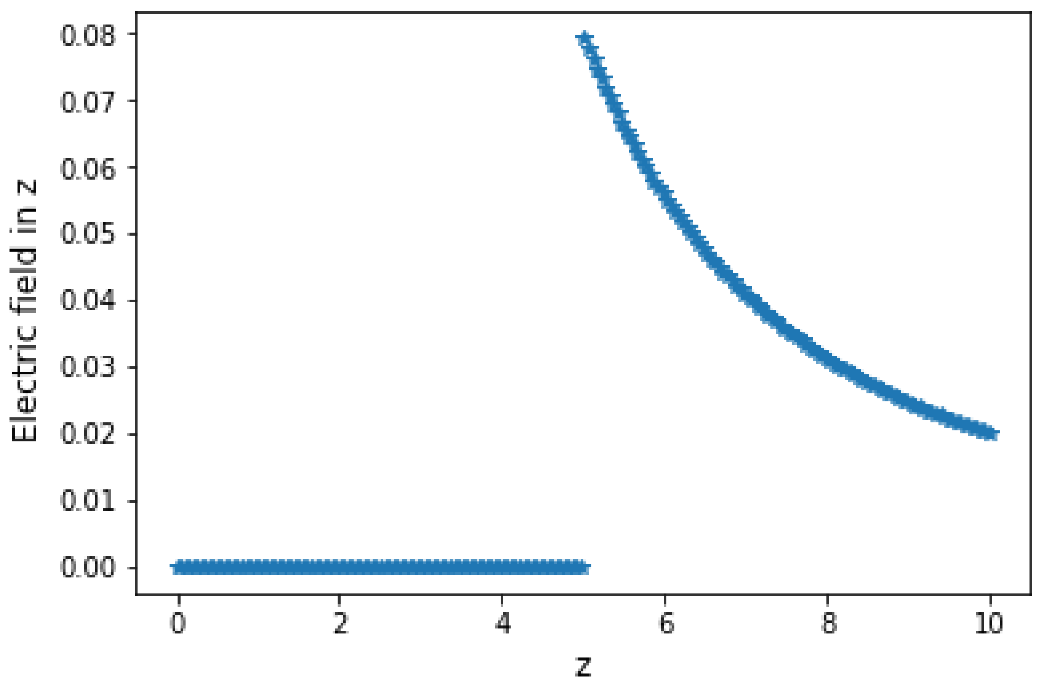
And this is the point where we will integrate using some computational methods.

[2]:
```
import scipy.integrate as integrate
import matplotlib.pyplot as plt
import numpy as np
import random
```

[3]:
```
function = lambda theta: (z-R*np.cos(theta))*np.sin(theta)/((R**2+z**2-2*R*z*np.
↪cos(theta))**(1.5))
```

[4]:
```
R=5
zz = np.linspace(0,10,200)
E_z=[]
error=[]
for i in range (len(zz)):
    z = zz[i]
    E_z = np.append(E_z, integrate.quad(function, 0, np.pi)[0])
    error = np.append(E_z, integrate.quad(function, 0 , np.pi)[1])
```

[5]:
```
plt.plot(zz, E_z,'*')
plt.xlabel("z",fontsize="large")
plt.ylabel("Electric field in z ",fontsize="large")
plt.show()
```



[11]:
```
#Now we will use a variable step size integrator to integrate the function.

import numpy as np
import sys
sys.setrecursionlimit(20000)

def simple_integrate(a,b,tol,z,R):
    u=np.linspace(a,b,5)
    y=(z-R*u)/(R**2+z**2-2*R*z*u)**(1.5)
    f1=(y[0]+4*y[2]+y[4])/6.0*(b-a)
    f2=(y[0]+4*y[1]+2*y[2]+4*y[3]+y[4])/12.0*(b-a)
    myerr=np.abs(f2-f1)
    neval=len(u)
    if (myerr<tol):
        #return (f2)/1.0,myerr,neval
        return (16.0*f2-f1)/15.0,myerr,neval
    else:
        mid=0.5*(b+a)
        f_left,err_left,neval_left=simple_integrate(a,mid,tol/2.0,2,3)
        f_right,err_right,neval_right=simple_integrate(mid,b,tol/2.0,2,3)
        neval=neval+neval_left+neval_right
        f=f_left+f_right
        err=err_left+err_right
        return f,err,neval
```

```
f,err,neval=simple_integrate(0,np.pi,1E-6,2,3);
#f,err,neval=simple_integrate(fun,-1,1,1e-4);pred=np.arctan(1)-np.arctan(-1)
#a=-5;b=5;f,err,neval=simple_integrate(fun2,a,b,1e-4);pred=(b-a)+np.sqrt(2*np.
 ↪pi)*sig
```

```
/home/matheus/.local/lib/python2.7/site-packages/ipykernel_launcher.py:9:
RuntimeWarning: invalid value encountered in power
  if __name__ == '__main__':
```

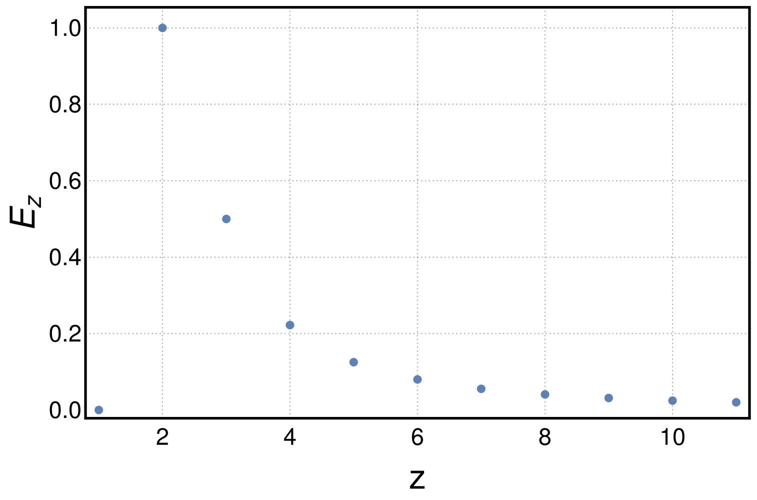
RuntimeError: maximum recursion depth exceeded.

We can see that quad does not care when we integrate, and the built integrator should compute it too, however, the maximum recursion depth was exceeded during the operation. The behavior for the electric field with an increasing r was obtained, decreasing within the $1/r^2$. Using Mathematica to integrate the function also does not seem to indicate any problem with $z = R$, as we can see in 1.

The fact that there is no singularity in this example is also supported by Gauss' law which states,

$$\nabla \cdot \boldsymbol{E} = \frac{\rho}{\epsilon_0}, \tag{13}$$

thus the divergence of the electric field should represent a physical quantity and not a singularity at a chosen given point for a simple case like the electric field of a sphere.



The complete program is shown in 2.

```
z = 1;
R = 1;
Integrate[(z - R u)/(R^2 + z^2 - 2 R z u)^(3/2), {u, -1, 1}]
```

```
1
```

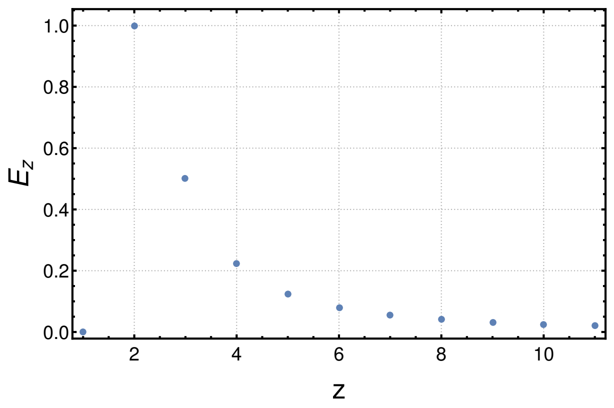
```
Integrate[(z - R Cos[θ]) Sin[θ]/(R^2 + z^2 - 2 R z Cos[θ])^(3/2), {θ, 0, π}]
```

$$\frac{2}{25}$$

```
Table[Integrate[(z - R Cos[θ]) Sin[θ]/(R^2 + z^2 - 2 R z Cos[θ])^(3/2), {θ, 0, π}], {z, 0, 10}]
```

$$\left\{0, 1, \frac{1}{2}, \frac{2}{9}, \frac{1}{8}, \frac{2}{25}, \frac{1}{18}, \frac{2}{49}, \frac{1}{32}, \frac{2}{81}, \frac{1}{50}\right\}$$

```
ListPlot[Table[Integrate[(z - R Cos[θ]) Sin[θ]/(R^2 + z^2 - 2 R z Cos[θ])^(3/2), {θ, 0, π}], {z, 0, 10, 1}], PlotTheme → "Detailed",
 LabelStyle → Directive[Black, FontFamily → "Helvetica", FontSize → 24], FrameStyle → Directive[Black, Thick],
 FrameLabel → {Style[HoldForm["z"], 26, Black], Style[HoldForm["Ez"], 26, Black]}, PlotRange → All, FrameTicksStyle → Directive[Black, 18], ImageSize → Large]
```



E_z
1.0
0.8
0.6
0.4
0.2
0.0
2
4
6
8
10
z