Siddhanth Kumar Section 102

Homework 6: Numerical Integration and Differentiation, Monte Carlo

Please complete this homework assignment in code cells in the iPython notebook. Include comments in your code when necessary. Please rename the notebook as SIS ID_HW06.ipynb (your student ID number) and save the notebook once you have executed it as a PDF (note, that when saving as PDF you don't want to use the option with latex because it crashes, but rather the one to save it directly as a PDF).

The homework should be submitted on bCourses under the Assignments tab (both the .ipynb and .pdf files). Please label your submission with your student ID number (SIS ID)

Problem 1: Numerical integration [Ayars 2.2]

Compare results of the trapezoid integration method, Simpson's method, and the adaptive Gaussian quadrature method for the following integrals:

1. $$\int_0^{\pi/2} \cos x \, dx$$
2. $$\int_1^3 \frac{1}{x^2} \, dx$$
3. $$\int_2^4 (x^2 + x + 1) \, dx$$
4. $$\int_0^{6.9} \cos\left(\frac{\pi}{2} x^2\right) dx$$

For each part, try it with more and with fewer slices to determine how many slices are required to give an 'acceptable' answer. (If you double the number of slices and still get the same answer, then try half as many, etc.) Parts (3) and (4) are particularly interesting in this regard. In your submitted work, describe roughly how many points were required, and explain.

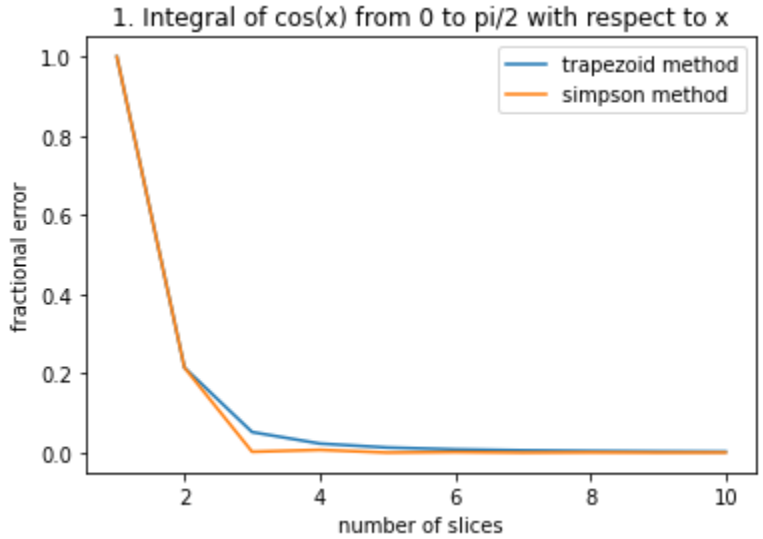
In [10]:
```
import numpy as np
import scipy.integrate
import matplotlib.pyplot as plt

def func1(x):
    return np.cos(x)

def func2(x):
```

```
    return (x**(-2))

def func3(x):
    return ((x*x) + x + 1)

def func4(x):
    return np.cos((np.pi / 2)*(x**2))

def calc(N,func, lowerBound, upperBound):

    trapError = []
    simpError = []

    for N in range(1, N):
        val1, err1 = scipy.integrate.quad(func, lowerBound, upperBound)
        x = np.linspace(lowerBound, upperBound, N)
        y= []

        for value in x:
            y.append(np.cos(value))


        integral_trapz = scipy.integrate.trapz(y, x)
        integral_simps = scipy.integrate.simps(y, x)
        #print('For N = {0}'.format(N))
        #print("Trapezoid rule gives:\t %.9f" % integral_trapz)
        #print("Simpson rule gives:\t %.9f" % integral_simps)
        #print(val1)
        #print('\n \n \n')

        trapError.append(np.abs(val1 - integral_trapz)/val1)
        simpError.append(np.abs(val1 - integral_simps)/val1)

    x_final = np.linspace(lowerBound, upperBound, N)
    y_final = []

    for value in x_final:
        y_final.append(np.cos(value))

    final_integral_trapz = scipy.integrate.trapz(y,x)
    final_integral_simps = scipy.integrate.simps(y,x)
    final_val1, final_err1 = scipy.integrate.quad(func, lowerBound, upperBound)

    return trapError, simpError, final_integral_trapz, final_integral_simps, final_val1

def calc1(N,func, lowerBound, upperBound):

    trapError = []
    simpError = []

    for N in range(1, N):
        val1, err1 = scipy.integrate.quad(func, lowerBound, upperBound)
        x = np.linspace(lowerBound, upperBound, N)
        y= []

        for value in x:
            y.append(value**(-2))


        integral_trapz = scipy.integrate.trapz(y, x)
        integral_simps = scipy.integrate.simps(y, x)
        #print('For N = {0}'.format(N))
        #print("Trapezoid rule gives:\t %.9f" % integral_trapz)
        #print("Simpson rule gives:\t %.9f" % integral_simps)
        #print(val1)
        #print('\n \n \n')
```

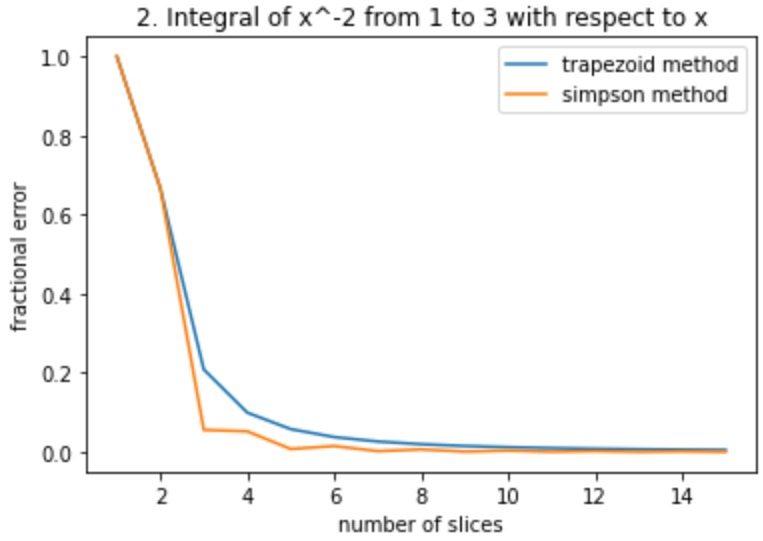
```
        trapError.append(np.abs(val1 - integral_trapz)/val1)
        simpError.append(np.abs(val1 - integral_simps)/val1)

    x_final = np.linspace(lowerBound, upperBound, N)
    y_final = []

    for value in x_final:
        y_final.append(value**(-2))

    final_integral_trapz = scipy.integrate.trapz(y,x)
    final_integral_simps = scipy.integrate.simps(y,x)
    final_val1, final_err1 = scipy.integrate.quad(func, lowerBound, upperBound)

    return trapError, simpError, final_integral_trapz, final_integral_simps, final_val1


def calc2(N,func, lowerBound, upperBound):

    trapError = []
    simpError = []

    for N in range(1, N):
        val1, err1 = scipy.integrate.quad(func, lowerBound, upperBound)
        x = np.linspace(lowerBound, upperBound, N)
        y= []

        for value in x:
            y.append((value*value) + value + 1)


        integral_trapz = scipy.integrate.trapz(y, x)
        integral_simps = scipy.integrate.simps(y, x)
        #print('For N = {0}'.format(N))
        #print("Trapezoid rule gives:\t %.9f" % integral_trapz)
        #print("Simpson rule gives:\t %.9f" % integral_simps)
        #print(val1)
        #print('\n \n \n')

        trapError.append(np.abs(val1 - integral_trapz)/val1)
        simpError.append(np.abs(val1 - integral_simps)/val1)


    x_final = np.linspace(lowerBound, upperBound, N)
    y_final = []

    for value in x_final:
        y_final.append((value*value) + value + 1)

    final_integral_trapz = scipy.integrate.trapz(y,x)
    final_integral_simps = scipy.integrate.simps(y,x)
    final_val1, final_err1 = scipy.integrate.quad(func, lowerBound, upperBound)

    return trapError, simpError, final_integral_trapz, final_integral_simps, final_val1



def calc3(N,func, lowerBound, upperBound):

    trapError = []
    simpError = []

    for N in range(1, N):
        val1, err1 = scipy.integrate.quad(func, lowerBound, upperBound)
        x = np.linspace(lowerBound, upperBound, N)
        y= []
```

```
        for value in x:
            y.append(np.cos((np.pi * (value*value))/2))


        integral_trapz = scipy.integrate.trapz(y, x)
        integral_simps = scipy.integrate.simps(y, x)
        #print('For N = {0}'.format(N))
        #print("Trapezoid rule gives:\t %.9f" % integral_trapz)
        #print("Simpson rule gives:\t %.9f" % integral_simps)
        #print(val1)
        #print('\n \n \n')

        trapError.append(np.abs(val1 - integral_trapz)/val1)
        simpError.append(np.abs(val1 - integral_simps)/val1)

    x_final = np.linspace(lowerBound, upperBound, N)
    y_final = []

    for value in x_final:
        y_final.append(np.cos((np.pi * (value*value))/2))

    final_integral_trapz = scipy.integrate.trapz(y,x)
    final_integral_simps = scipy.integrate.simps(y,x)
    final_val1, final_err1 = scipy.integrate.quad(func, lowerBound, upperBound)

    return trapError, simpError, final_integral_trapz, final_integral_simps, final_val1



trapError, simpError, final_integral_trapz, final_integral_simps, final_val = calc(11,fu
xs = np.linspace(1, 10, 10)

trapError1, simpError1, final_integral_trapz1, final_integral_simps1, final_val1 = calc1
xs1 = np.linspace(1, 15, 15)

trapError2, simpError2, final_integral_trapz2, final_integral_simps2, final_val2 = calc2
xs2= np.linspace(1, 10, 10)

trapError3, simpError3, final_integral_trapz3, final_integral_simps3, final_val3 = calc3
xs3= np.linspace(1, 100, 100)



plt.plot(xs, trapError)
plt.plot(xs, simpError)
plt.xlabel('number of slices')
plt.ylabel('fractional error')
plt.legend(['trapezoid method', 'simpson method'])
plt.title('1. Integral of cos(x) from 0 to pi/2 with respect to x')
plt.show()
print('The trapezoidal method gives {0:.3f}'.format(final_integral_trapz))
print('Simpson method gives {0:.3f}'.format(final_integral_simps))
print('Adaptive gaussian quadrative function gives {0:.3f}'.format(final_val))


plt.plot(xs1, trapError1)
plt.plot(xs1, simpError1)
plt.xlabel('number of slices')
plt.ylabel('fractional error')
plt.legend(['trapezoid method', 'simpson method'])
plt.title('2. Integral of x^-2 from 1 to 3 with respect to x')
plt.show()
print('The trapezoidal method gives {0:.3f}'.format(final_integral_trapz1))
print('Simpson method gives {0:.3f}'.format(final_integral_simps1))
```

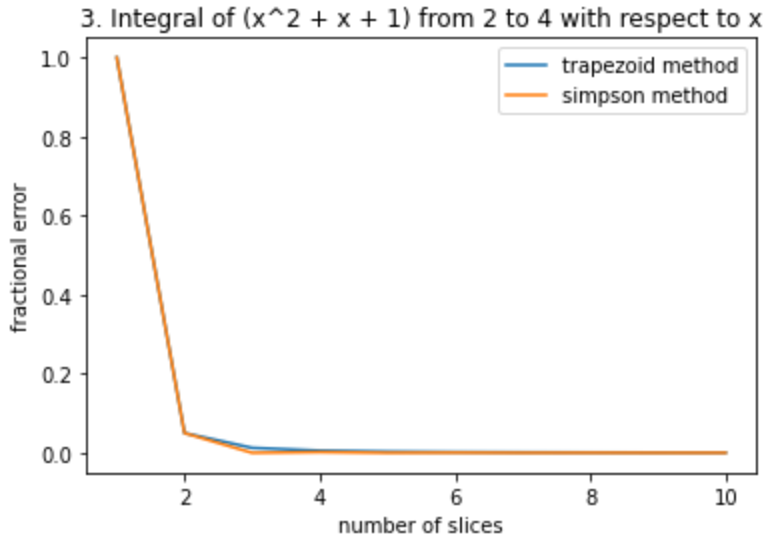
```
print('Adaptive gaussian quadrative function gives {0:.3f}'.format(final_val1))


plt.plot(xs2, trapError2)
plt.plot(xs2, simpError2)
plt.xlabel('number of slices')
plt.ylabel('fractional error')
plt.legend(['trapezoid method', 'simpson method'])
plt.title('3. Integral of (x^2 + x + 1) from 2 to 4 with respect to x')
plt.show()
print('The trapezoidal method gives {0:.3f}'.format(final_integral_trapz2))
print('Simpson method gives {0:.3f}'.format(final_integral_simps2))
print('Adaptive gaussian quadrative function gives {0:.3f}'.format(final_val2))

plt.plot(xs3, trapError3)
plt.plot(xs3, simpError3)
plt.xlabel('number of slices')
plt.ylabel('fractional error')
plt.legend(['trapezoid method', 'simpson method'])
plt.title('4. Integral of cos(pi * x^2 / 2) from 0 to 6.9 with respect to x')
plt.show()
print('The trapezoidal method gives {0:.3f}'.format(final_integral_trapz3))
print('Simpson method gives {0:.3f}'.format(final_integral_simps3))
print('Adaptive gaussian quadrative function gives {0:.3f}'.format(final_val3))




'''The best way to "explain" how many slices/points are needed to get an acceptable answ
   make plots of the fractional error vs number of slices.  The fractional error is defi
   abs(estimate - exact)/exact, where "estimate" is the value of the integral using one
   integration methods and "exact" is the analytical value of the integral (by hand or u
   If you do everything correctly, you should find that the fractional error approaches
   the number of slices.'''
```



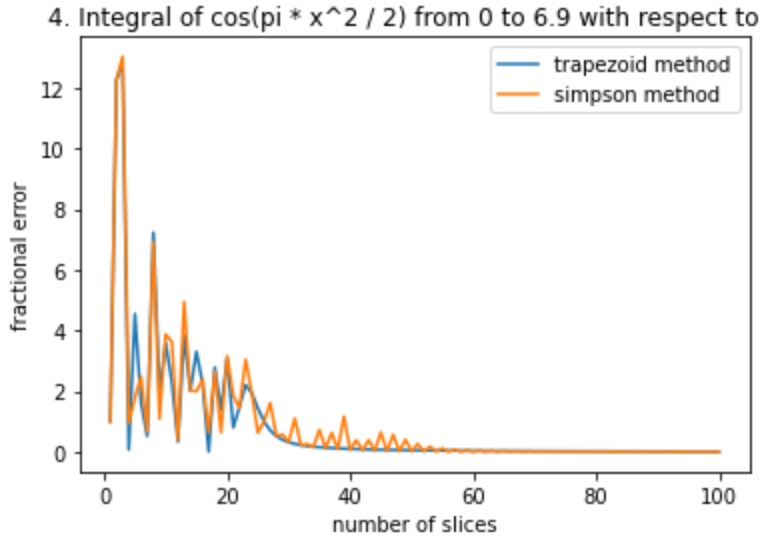

```
The trapezoidal method gives 0.997
Simpson method gives 1.000
Adaptive gaussian quadrative function gives 1.000
```




```
The trapezoidal method gives 0.670
Simpson method gives 0.667
Adaptive gaussian quadrative function gives 0.667
```




```
The trapezoidal method gives 26.683
Simpson method gives 26.668
Adaptive gaussian quadrative function gives 26.667
```




```
The trapezoidal method gives 0.478
Simpson method gives 0.471
Adaptive gaussian quadrative function gives 0.473
```

Out[10]:
```
'The best way to "explain" how many slices/points are needed to get an acceptable answer
is to just\n   make plots of the fractional error vs number of slices.  The fractional e
rror is defined as \n   abs(estimate - exact)/exact, where "estimate" is the value of th
e integral using one of these three\n   integration methods and "exact" is the analytica
```

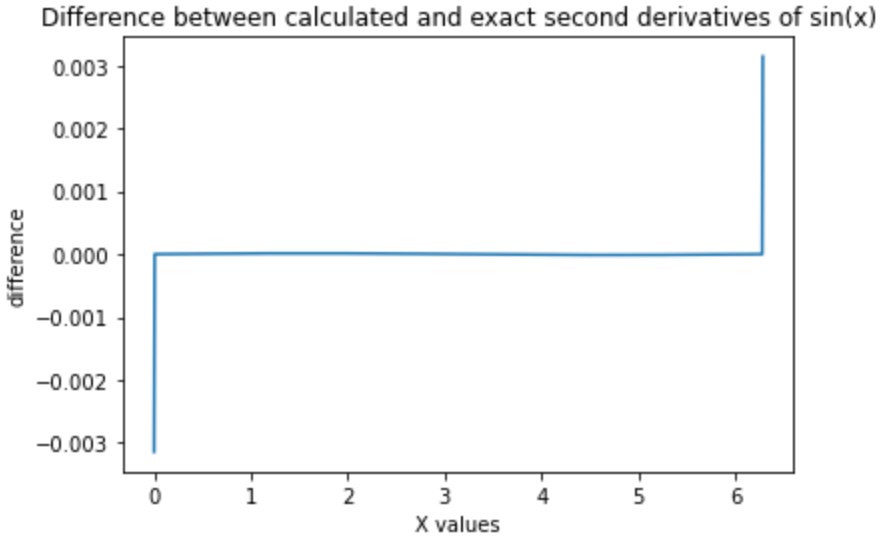
```
l value of the integral (by hand or using WolframAlpha).\n   If you do everything correc
tly, you should find that the fractional error approaches 0 as you increase\n   the numb
er of slices.'
```

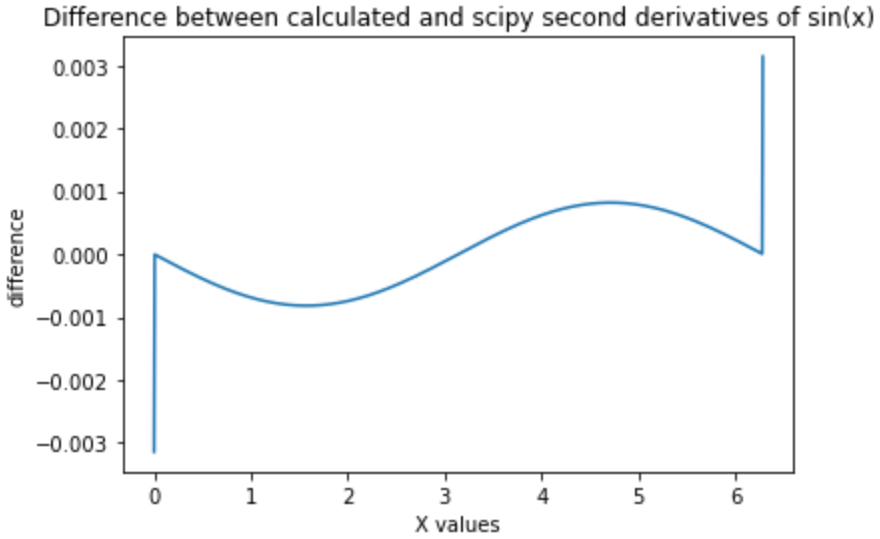
Problem 2: Numerical differentiation [Ayars 2.8]

Write a function that, given a list of x-values x_i and function values $f_i(x_i)$, returns a list of values of the second derivative $f''(x_i)$ of the function. Test your function by giving it a list of known function values for $\sin(x)$ and making a graph of the differences between the output of the function and $-\sin(x)$. Compare your output to Python's *scipy.misc.derivative*

In [16]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.misc import derivative

x_values = np.linspace(0, 2*np.pi, 1000)
function_values = np.sin(x_values)
#plt.plot(x_values, function_values)
#plt.show()

def centered_dy(y, x):
    '''
        Uses centered differences (see below) to estimate the derivatives at each value
        except for the first and last values. The derivative at the first value of x is
        using a forward difference. The derivative at the last value of x is estimated
        using a backward difference.
            dy/dx at x[i] is approximated by (y[i+1] - y[i-1]) / (x[i+1]-x[i-1])
    '''
    dyc = [0.0]*len(x)
    dyc[0] = (y[0] - y[1])/(x[0] - x[1])
    for i in range(1,len(y)-1):
        dyc[i] = (y[i+1] - y[i-1])/(x[i+1]-x[i-1])
    dyc[-1] = (y[-1] - y[-2])/(x[-1] - x[-2])

    dyc1 = [0.0]*len(x)
    dyc1[0] = (dyc[0] - dyc[1])/(x[0] - x[1])
    for i in range(1, len(dyc) - 1):
        dyc1[i] = (dyc[i+1] - dyc[i-1])/(x[i+1]-x[i-1])
    dyc1[-1] = (dyc[-1] - dyc[-2])/(x[-1] - x[-2])

    return dyc1


secondDeriv = centered_dy(function_values, x_values)

plt.plot(x_values, secondDeriv + np.sin(x_values))
plt.title('Difference between calculated and exact second derivatives of sin(x)')
plt.xlabel('X values')
plt.ylabel('difference')
plt.show()
plt.plot(x_values, secondDeriv - derivative(np.sin, x_values, dx=0.1, n=2))
plt.title('Difference between calculated and scipy second derivatives of sin(x)')
plt.xlabel('X values')
plt.ylabel('difference')
plt.show()


def second_derivative(x_values, function_values):
    '''Write your function to calculate and return
       the values of the second derivative.  You can think
       of it as two first-order derivatives, or see
       "higher order differences" on this wiki page:
```

```
        https://en.wikipedia.org/wiki/Finite_difference'''

# Compare your second derivative values to both the exact answer and scipy.misc.derivati
```





Problem 3: MC integration [similar to Ayars 6.2, Newman 10.7]

The "volume" of a 2-sphere $x^2 + y^2 \leq r^2$ (aka "circle") is $(1)\pi r^2$. The volume of a 3-sphere $x^2 + y^2 + z^2 \leq r^2$ is $(\frac{4}{3})\pi r^3$. The equation for an N-sphere is $x_1^2 + x_2^2 + x_3^2 + \ldots + x_N^2 \leq r^2$ (where x_i are spatial coordinates in N dimensions). We can guess, by induction from the 2-dimensional and 3-dimensional cases, that the "volume" of an N-sphere is $\alpha_N \pi r^N$. Write a function that uses Monte Carlo integration to estimate α_N and its uncertainty for a fixed N. Graph α_N with its uncertainty as a function of N for $N = 4 \ldots 10$.

First, here's the standard import statements and some plotting parameters.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
plt.rcParams['figure.figsize'] = 10, 5
plt.rcParams['font.size'] = 14
```

If you're having trouble getting started, we did the $N = 2$ case in Workshop 8. To use this method of MC integration, start by generating a large number (something like 10000) of points randomly scattered in some

region of $N-$dimensional space. I say "some region", because you may choose to sample points from the range $(0, 1)$ or the range $(-1, 1)$ along each dimension. I prefer to use `np.random.rand()`, which samples uniformly from the range $(0, 1)$ and scale things appropriately (this choice of range required us to multiply by 4 to estimate π in WS8; and of course this scale factor depends on N !).

Then you just need to count up the number of these random points that satisfy the $N-$sphere condition given above (it's easiest just to take $r = 1$). The fraction of points within the $N-$sphere can be used to estimate α_N. You could repeat this procedure many times to estimate the uncertainty in each α_N or you may find it faster to use an analytical formula for the error (see lecture notes on statistics `lec05_stat.pdf` or MC `lec06_MC.pdf` in bCourses).

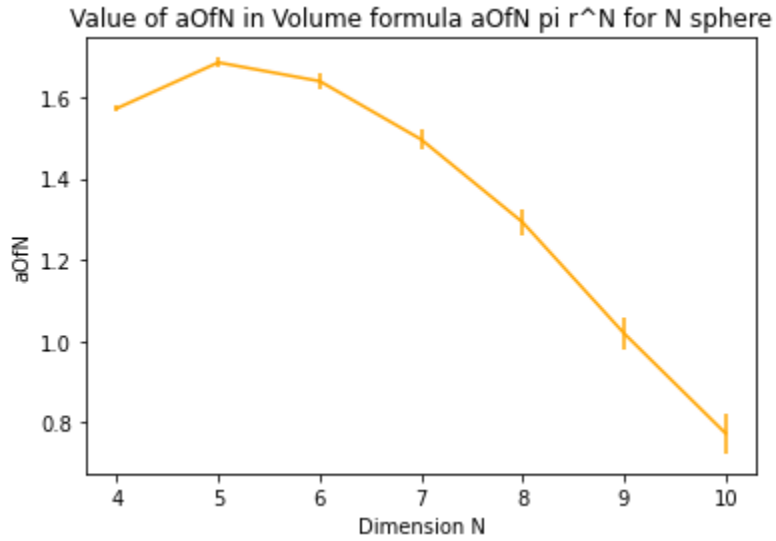
In [15]:
```
# It's probably easiest to write a function that finds alpha and its error for a given N
# Then go through values of N from 4 to 10, and get the alpha estimates from this functi
# Finally, plot these estimates (along with error bars) -- plt.errorbar() is helpful for

from random import uniform

def generate_point(dimensions):
    point = [uniform(-1, 1) for dimension in range(dimensions)]
    return point

def find_dist(point):
    ret = 0
    for dim in point:
        ret += dim**2
    return ret

def total_area(dimensions):
    return 2**dimensions

def find_area(dimensions, total_points):
    in_count = 0

    for point in range(total_points):
        if find_dist(generate_point(dimensions)) <= 1:
            in_count += 1

    sigma = (total_area(dimensions))/np.pi * np.sqrt(in_count * (1 - in_count/total_poin

    return in_count*total_area(dimensions)/total_points, sigma


N = [4, 5, 6, 7, 8, 9, 10]
areas = []
sigmas = []

for value in N:
    sol, sigma = find_area(value, 100000)
    print(sigma)
    sol = sol/np.pi
    sigma=sigma
    areas.append(sol)
    sigmas.append(sigma)
    print('For N = {0}, a = {1}'.format(value, sol))


plt.errorbar(N, areas, yerr=sigmas, color='orange')
plt.title('Value of aOfN in Volume formula aOfN pi r^N for N sphere')
plt.xlabel('Dimension N')
plt.ylabel('aOfN')
```

```
0.007441178708873714
```

```
For N = 4, a = 1.5731129223111882
0.011971659134871457
For N = 5, a = 1.6861765938836706
0.01752784482780331
For N = 6, a = 1.6401362519460472
0.02423508674618243
For N = 7, a = 1.4965148312999208
0.032193643168243594
For N = 8, a = 1.2923890674879794
0.04061614190923464
For N = 9, a = 1.0185916357881302
0.050119762825405305
For N = 10, a = 0.772499896581718
```

Out[15]:

```
Text(0, 0.5, 'aOfN')
```



In [ ]: