Homework06

April 19, 2021

Victor Cruz Ramos Section 101

1 Homework 6: Numerical Integration and Differentiation, Monte Carlo

**** Submit this notebook to bourses to receive a credit for this assignment. **** Please complete this homework assignment in code cells in the iPython notebook. Please submit both a PDF of the jupyter notebook to bcourses and the notebook itself (.ipynb file). 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.

1.1 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.

[15]:
```
import numpy as np
import scipy.integrate
import matplotlib.pyplot as plt
```

```
'''The best way to "explain" how many slices/points are needed to get an␣
↪acceptable answer is to just
  make plots of the fractional error vs number of slices.  The fractional␣
↪error is defined as
  abs(estimate - exact)/exact, where "estimate" is the value of the integral␣
↪using one of these three
  integration methods and "exact" is the analytical value of the integral (by␣
↪hand or using WolframAlpha).
  If you do everything correctly, you should find that the fractional error␣
↪approaches 0 as you increase
  the number of slices.'''

def func_1(x):
    return np.cos(x)
N_1 = np.linspace(0, 2*np.pi, 5) # 5 was the lowest amount of points I could get
y_1 = np.cos(N_1)
integral_trapz = scipy.integrate.trapz(func_1(N_1))
integral_quad = scipy.integrate.quad(func_1, 0, 2*np.pi)
integral_simps = scipy.integrate.simps(y_1, N_1)
#print(integral_quad)

print("Trapezoid rule gives:\t %.5f" % integral_trapz)
print("Simpson rule gives:\t %.5f" % integral_simps)
print("Exact value:\t\t %.5f" % integral_quad[0])
```

```
Trapezoid rule gives:    -0.00000
Simpson rule gives:      -0.00000
Exact value:             0.00000
```

[16]:
```
def func_2(x):
    return 1/(x)**2
N_2 = np.linspace(1, 3, 5) # I found here that once again the lowest amount of␣
↪points is 5
y_2 = 1/(N_2)**2
integral_trapz_2 = scipy.integrate.trapz(func_2(N_2))
integral_quad_2 = scipy.integrate.quad(func_2, 1, 3)
integral_simps_2 = scipy.integrate.simps(y_2, N_2)

print("Trapezoid rule gives:\t %.5f" % integral_trapz_2)# trapezoid rule def is␣
↪very innaccurate here
print("Simpson rule gives:\t %.5f" % integral_simps_2)
print("Exact value:\t\t %.5f" % integral_quad_2[0])
```

```
Trapezoid rule gives:    1.41000
Simpson rule gives:      0.67148
Exact value:             0.66667
```

[25]:
```
def func_3(x):
    return ((x)**2+x+1)
N_3 = np.linspace(2, 4, 3) # the least number of points I could do was 3 here
y_3 = (N_3)**2 + N_3 +1

integral_trapz_3 = scipy.integrate.trapz(func_3(N_3))
integral_quad_3 = scipy.integrate.quad(func_3, 2, 4)
integral_simps_3 = scipy.integrate.simps(y_3, N_3)

print("Trapezoid rule gives:\t %.5f" % integral_trapz_3) #trapezoid rule def is␣
 ↪very innaccurate here
print("Simpson rule gives:\t %.5f" % integral_simps_3)
print("Exact value:\t\t %.5f" % integral_quad_3[0])
```

```
Trapezoid rule gives:    27.00000
Simpson rule gives:      26.66667
Exact value:             26.66667
```

[40]:
```
def func_4(x):
    return np.cos(np.pi/2*x**2)
N_4 = np.linspace(0, 6.9, 4) # the min amount of points here I found to be 4
y_4 = np.cos(np.pi/2*(N_4)**2)

integral_trapz_4 = scipy.integrate.trapz(func_4(N_4))
integral_quad_4 = scipy.integrate.quad(func_4, 0, 6.9)
integral_simps_4 = scipy.integrate.simps(y_4, N_4)

print("Trapezoid rule gives:\t %.5f" % integral_trapz_4) #trapezoid rule def is␣
 ↪very innaccurate here
print("Simpson rule gives:\t %.5f" % integral_simps_4)
print("Exact value:\t\t %.5f" % integral_quad_4[0])
```

```
Trapezoid rule gives:    0.22045
Simpson rule gives:      0.02656
Exact value:             0.47323
```

1.2 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*

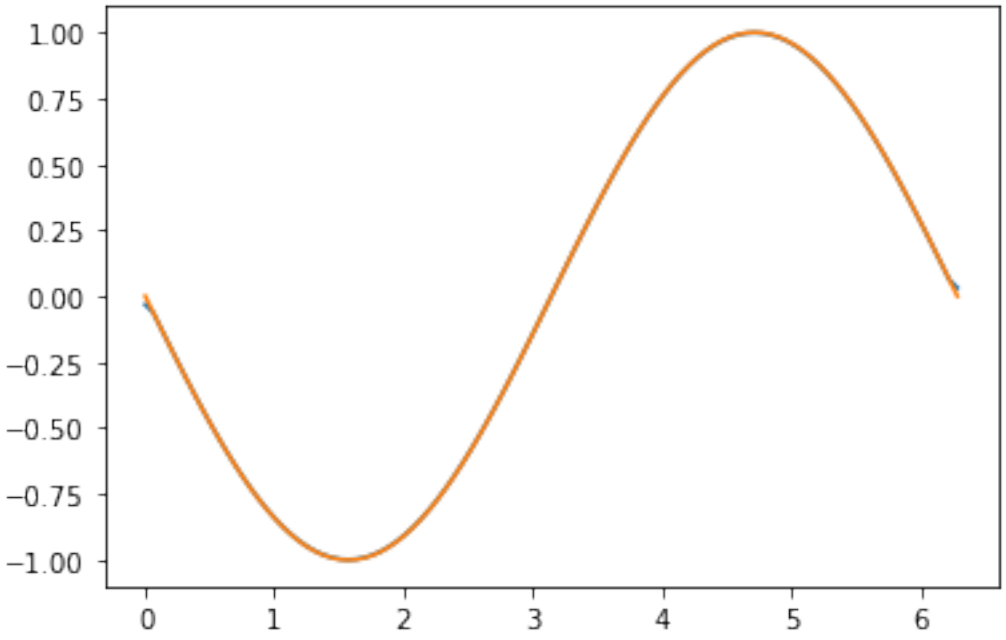
[58]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.misc import derivative
```

```
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'''
    deriv = [0.0]*len(x_values)
    deriv[0] = (function_values[0] - function_values[1])/(x_values[0] - x_values[1])
    for i in range(1,len(function_values)-1):
        deriv[i] = (function_values[i+1] - function_values[i-1])/(x_values[i+1]-x_values[i-1])
    deriv[-1] = (function_values[-1] - function_values[-2])/((x_values[-1] - x_values[-2]))

    deriv_2 = [0.0]*len(deriv)
    deriv_2[0] = (deriv[0]-deriv[1])/(x_values[0] - x_values[1])
    for i in range(1,len(deriv)-1):
        deriv_2[i] = (deriv[i+1] - deriv[i-1])/(x_values[i+1]-x_values[i-1])
    deriv_2[-1] = (deriv[-1] - deriv[-2])/((x_values[-1] - x_values[-2]))

    return deriv_2

def f(x):
    return np.sin(x)
x = np.linspace(0, 2*np.pi, 101)

print(second_derivative(x, f(x)))
scipy.misc.derivative(func=np.sin, x0=2*np.pi, dx=1e-6, n=2, order=5 )

plt.plot(x, second_derivative(x, f(x)))
plt.plot(x, -np.sin(x))

# Compare your second derivative values to both the exact answer and scipy.misc.derivative (with n = 2)
```

```
[-0.03138493249728743, -0.06270793399949973, -0.12516838843203507,
-0.18713486041942407, -0.2483627966065229, -0.3086105583015837,
-0.3676403751137834, -0.4252192833243225, -0.48112004528778957,
-0.535122046235349, -0.5870121649404185, -0.6365856148108152,
-0.683646752087872, -0.7280098479629198, -0.7694998215641131,
-0.8079529309206289, -0.8432174191773846, -0.8751541135100525,
-0.903636974376573, -0.9285535929376838, -0.9498056346832494,
-0.967309227513643, -0.9809952927446147, -0.9908098177292286,
-0.9967140690211946, -0.9986847452380966, -0.9967140690212014,
-0.9908098177292118, -0.9809952927446061, -0.9673092275136589,
-0.94980563468326, -0.9285535929376777, -0.9036369743765577,
```

```
-0.8751541135100475, -0.8432174191773943, -0.8079529309206348,
-0.7694998215641203, -0.7280098479629148, -0.6836467520878575,
-0.6365856148108231, -0.5870121649404263, -0.5351220462353462,
-0.4811200452877843, -0.4252192833243176, -0.3676403751137866,
-0.3086105583015888, -0.2483627966065216, -0.1871348604194229,
-0.1251683884320305, -0.0627079339995005, 1.7669748230352908e-15,
0.06270793399950163, 0.1251683884320305, 0.18713486041942293,
0.24836279660652782, 0.30861055830158857, 0.3676403751137835,
0.4252192833243208, 0.48112004528778873, 0.5351220462353444, 0.5870121649404135,
0.6365856148108222, 0.6836467520878741, 0.7280098479629095, 0.7694998215641178,
0.8079529309206372, 0.8432174191773896, 0.8751541135100466, 0.9036369743765634,
0.9285535929376866, 0.9498056346832491, 0.9673092275136559, 0.9809952927446147,
0.9908098177292136, 0.9967140690211999, 0.9986847452380904, 0.9967140690211996,
0.9908098177292243, 0.9809952927446077, 0.9673092275136622, 0.9498056346832497,
0.9285535929376734, 0.90363697437657, 0.8751541135100536, 0.8432174191773825,
0.8079529309206266, 0.7694998215641276, 0.7280098479629139, 0.6836467520878637,
0.6365856148108261, 0.5870121649404162, 0.5351220462353412, 0.48112004528778607,
0.4252192833243246, 0.3676403751137884, 0.3086105583015833, 0.2483627966065225,
0.18713486041942293, 0.12516838843203404, 0.06270793399950009,
0.03138493249728761]
```

[58]: [<matplotlib.lines.Line2D at 0x7f512cd7e220>]



1.3 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 $x_1^2 + x_2^2 + x_3^2 + ... + x_N^2 \leq r^2$ (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 $N = 4$ … 10.

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

[59]:
```
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).

[60]:
```
# 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 function
# Finally, plot these estimates (along with error bars) -- plt.errorbar() is␣
↪helpful for this
```

[95]:
```
import random

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

def monte_carlo(N, n):
    #x**2+y**2=1
    circle_points= 0

    for i in range(n):
        x = np.random.rand(N)

        xsquaredvals = [i**2 for i in x]
```

```
        x_vals =␣
↪xsquaredvals[0]+xsquaredvals[1]+xsquaredvals[2]+xsquaredvals[3]#+xsquaredvals[4]+xsquaredva

        if x_vals <= 1:
            circle_points += 1
        print(x_vals)
    alpha = ((circle_points/n)*(2**N))/np.pi
    return alpha



monte_carlo(N[0], 10)
```

```
0.3343050464456382
2.767416473059595
1.4468648471888668
1.1699527736443056
0.930078994899759
1.55378976949841
2.6321422230090255
0.22828618438931944
1.2076395840862766
0.8860089987939523
```

[95]: 2.0371832715762603

[96]:
```
alpha_val = []
errors = []
for i in range(N):
    alpha, error = monte_carlo(i, 1000)
    alpha_val.append(alpha)
    errors.append(error)
print(error)
print(alpha_val) # I feel that I was very close, but I couldn't figure out the␣
↪final step. I tried my best lol
# some code I used after attending office hours today

#monte_carlo(x_cor, y_cor)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-96-8b7db9fda72f> in <module>
      1 alpha_val = []
      2 errors = []
----> 3 for i in range(N):
      4     alpha, error = monte_carlo(i, 1000)
      5     alpha_val.append(alpha)
```

```
TypeError: 'list' object cannot be interpreted as an integer
```

[ ]: