3038741162_Homework06

April 15, 2023

1 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)

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.

[1]:
```
import numpy as np
import scipy.integrate
import matplotlib.pyplot as plt
```

```python
'''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 fractional_error(estimate, exact):
    return np.abs((estimate - exact) / exact)

n_slices = [2, 4, 8, 16, 32, 64, 128, 256, 512]

#1.
def f1(x):
    return np.cos(x)

a1 = 0
b1 = np.pi/20

exact = 1
print(f'Problem 1) Exact value: {exact:.8f}\n')

for n in n_slices:
    x = np.linspace(a1, b1, n)
    y = f1(x)

    integral_trapz = scipy.integrate.trapz(y, x)
    integral_simps = scipy.integrate.simps(y, x)
    integral_gauss, _ = scipy.integrate.quad(f1, a1, b1)

    f_error_trapz = fractional_error(integral_trapz, exact)
    f_error_simps = fractional_error(integral_simps, exact)
    f_error_gauss = fractional_error(integral_gauss, exact)

    print(f'Number of slices: {n}')
    print('trapz: {:.8f}, simps: {:.8f}, quad: {:.8f}'.format(integral_trapz,␣
↪integral_simps, integral_gauss))
    print('[fractional error] trapz: {:.8f}, simps: {:.8f}, quad: {:.8f}'.
↪format(f_error_trapz, f_error_simps, f_error_gauss))
```

```
#2.
def f2(x):
    return 1 / x**2

a2 = 1
b2 = 3

exact = 2/3
print('')
print(f'Problem 2) Exact value: {exact:.8f}\n')

for n in n_slices:
    x = np.linspace(a2, b2, n)
    y = f2(x)

    integral_trapz = scipy.integrate.trapz(y, x)
    integral_simps = scipy.integrate.simps(y, x)
    integral_gauss, _ = scipy.integrate.quad(f2, a2, b2)

    f_error_trapz = fractional_error(integral_trapz, exact)
    f_error_simps = fractional_error(integral_simps, exact)
    f_error_gauss = fractional_error(integral_gauss, exact)

    print(f'Number of slices: {n}')
    print('trapz: {:.8f}, simps: {:.8f}, quad: {:.8f}'.format(integral_trapz,␣
 ↪integral_simps, integral_gauss))
    print('[fractional error] trapz: {:.8f}, simps: {:.8f}, quad: {:.8f}'.
 ↪format(f_error_trapz, f_error_simps, f_error_gauss))

#3.
def f3(x):
    return x**2 + x + 1

a3 = 4
b3 = 2

exact = 28
print('')
print(f'Problem 3) Exact value: {exact:.8f}\n')

for n in n_slices:
    x = np.linspace(a3, b3, n)
    y = f3(x)

    integral_trapz = scipy.integrate.trapz(y, x)
    integral_simps = scipy.integrate.simps(y, x)
    integral_gauss, _ = scipy.integrate.quad(f3, a3, b3)
    f_error_trapz = fractional_error(integral_trapz, exact)
    f_error_simps = fractional_error(integral_simps, exact)
    f_error_gauss = fractional_error(integral_gauss, exact)

    print(f'Number of slices: {n}')
    print('trapz: {:.8f}, simps: {:.8f}, quad: {:.8f}'.format(integral_trapz,␣
 ↪integral_simps, integral_gauss))
    print('[fractional error] trapz: {:.8f}, simps: {:.8f}, quad: {:.8f}'.
 ↪format(f_error_trapz, f_error_simps, f_error_gauss))

#4.
def f4(x):
    return np.cos((np.pi/2) * x**2)

a4 = 6.9
b4 = 0

exact = 1.5246
print('')
print(f'Problem 4) Exact value: {exact:.8f}\n')

for n in n_slices:
    x = np.linspace(a4, b4, n)
    y = f4(x)

    integral_trapz = scipy.integrate.trapz(y, x)
    integral_simps = scipy.integrate.simps(y, x)
    integral_gauss, _ = scipy.integrate.quad(f4, a4, b4)

    f_error_trapz = fractional_error(integral_trapz, exact)
    f_error_simps = fractional_error(integral_simps, exact)
    f_error_gauss = fractional_error(integral_gauss, exact)

    print(f'Number of slices: {n}')
    print('trapz: {:.8f}, simps: {:.8f}, quad: {:.8f}'.format(integral_trapz,␣
 ↪integral_simps, integral_gauss))
    print('[fractional error] trapz: {:.8f}, simps: {:.8f}, quad: {:.8f}'.
 ↪format(f_error_trapz, f_error_simps, f_error_gauss))
```

```
Problem 1) Exact value: 1.00000000

Number of slices: 2
trapz: 0.15611268, simps: 0.15611268, quad: 0.15643447
[fractional error] trapz: 0.84388732, simps: 0.84388732, quad: 0.84356553
Number of slices: 4
trapz: 0.15639872, simps: 0.15642256, quad: 0.15643447
```

```
[fractional error] trapz: 0.84360128, simps: 0.84357744, quad: 0.84356553
Number of slices: 8
trapz: 0.15642790, simps: 0.15643353, quad: 0.15643447
[fractional error] trapz: 0.84357210, simps: 0.84356647, quad: 0.84356553
Number of slices: 16
trapz: 0.15643304, simps: 0.15643437, quad: 0.15643447
[fractional error] trapz: 0.84356696, simps: 0.84356563, quad: 0.84356553
Number of slices: 32
trapz: 0.15643413, simps: 0.15643445, quad: 0.15643447
[fractional error] trapz: 0.84356587, simps: 0.84356555, quad: 0.84356553
Number of slices: 64
trapz: 0.15643438, simps: 0.15643446, quad: 0.15643447
[fractional error] trapz: 0.84356562, simps: 0.84356554, quad: 0.84356553
Number of slices: 128
trapz: 0.15643445, simps: 0.15643446, quad: 0.15643447
[fractional error] trapz: 0.84356555, simps: 0.84356554, quad: 0.84356553
Number of slices: 256
trapz: 0.15643446, simps: 0.15643447, quad: 0.15643447
[fractional error] trapz: 0.84356554, simps: 0.84356553, quad: 0.84356553
Number of slices: 512
trapz: 0.15643446, simps: 0.15643447, quad: 0.15643447
[fractional error] trapz: 0.84356554, simps: 0.84356553, quad: 0.84356553

Problem 2) Exact value: 0.66666667

Number of slices: 2
trapz: 1.11111111, simps: 1.11111111, quad: 0.66666667
[fractional error] trapz: 0.66666667, simps: 0.66666667, quad: 0.00000000
Number of slices: 4
trapz: 0.73281935, simps: 0.70129504, quad: 0.66666667
[fractional error] trapz: 0.09922902, simps: 0.05194255, quad: 0.00000000
Number of slices: 8
trapz: 0.67955867, simps: 0.67073870, quad: 0.66666667
[fractional error] trapz: 0.01933801, simps: 0.00610805, quad: 0.00000000
Number of slices: 16
trapz: 0.66950953, simps: 0.66716645, quad: 0.66666667
[fractional error] trapz: 0.00426429, simps: 0.00074968, quad: 0.00000000
Number of slices: 32
trapz: 0.66733412, simps: 0.66672874, quad: 0.66666667
[fractional error] trapz: 0.00100118, simps: 0.00009311, quad: 0.00000000
Number of slices: 64
trapz: 0.66682838, simps: 0.66667441, quad: 0.66666667
[fractional error] trapz: 0.00024257, simps: 0.00001161, quad: 0.00000000
Number of slices: 128
trapz: 0.66670647, simps: 0.66666763, quad: 0.66666667
[fractional error] trapz: 0.00005970, simps: 0.00000145, quad: 0.00000000
Number of slices: 256
trapz: 0.66667654, simps: 0.66666679, quad: 0.66666667
```

```
[fractional error] trapz: 0.00001481, simps: 0.00000018, quad: 0.00000000
Number of slices: 512
trapz: 0.66666913, simps: 0.66666668, quad: 0.66666667
[fractional error] trapz: 0.00000369, simps: 0.00000002, quad: 0.00000000

Problem 3) Exact value: 28.00000000

Number of slices: 2
trapz: -28.00000000, simps: -28.00000000, quad: -26.66666667
[fractional error] trapz: 2.00000000, simps: 2.00000000, quad: 1.95238095
Number of slices: 4
trapz: -26.81481481, simps: -26.71604938, quad: -26.66666667
[fractional error] trapz: 1.95767196, simps: 1.95414462, quad: 1.95238095
Number of slices: 8
trapz: -26.69387755, simps: -26.67055394, quad: -26.66666667
[fractional error] trapz: 1.95335277, simps: 1.95251978, quad: 1.95238095
Number of slices: 16
trapz: -26.67259259, simps: -26.66706173, quad: -26.66666667
[fractional error] trapz: 1.95259259, simps: 1.95239506, quad: 1.95238095
Number of slices: 32
trapz: -26.66805411, simps: -26.66671142, quad: -26.66666667
[fractional error] trapz: 1.95243050, simps: 1.95238255, quad: 1.95238095
Number of slices: 64
trapz: -26.66700260, simps: -26.66667200, quad: -26.66666667
[fractional error] trapz: 1.95239295, simps: 1.95238114, quad: 1.95238095
Number of slices: 128
trapz: -26.66674933, simps: -26.66666732, quad: -26.66666667
[fractional error] trapz: 1.95238390, simps: 1.95238098, quad: 1.95238095
Number of slices: 256
trapz: -26.66668717, simps: -26.66666675, quad: -26.66666667
[fractional error] trapz: 1.95238168, simps: 1.95238096, quad: 1.95238095
Number of slices: 512
trapz: -26.66667177, simps: -26.66666668, quad: -26.66666667
[fractional error] trapz: 1.95238113, simps: 1.95238095, quad: 1.95238095

Problem 4) Exact value: 1.52460000

Number of slices: 2
trapz: -6.27261653, simps: -6.27261653, quad: -0.47322531
[fractional error] trapz: 5.11427032, simps: 5.11427032, quad: 1.31039310
Number of slices: 4
trapz: -0.50702534, simps: -0.02655941, quad: -0.47322531
[fractional error] trapz: 1.33256286, simps: 1.01742058, quad: 1.31039310
Number of slices: 8
trapz: -3.89230316, simps: -3.74387608, quad: -0.47322531
[fractional error] trapz: 3.55299958, simps: 3.45564482, quad: 1.31039310
Number of slices: 16
trapz: 0.59607951, simps: 0.65549870, quad: -0.47322531
```

```
[fractional error] trapz: 0.60902564, simps: 0.57005201, quad: 1.31039310
Number of slices: 32
trapz: -0.57876380, simps: -0.57471732, quad: -0.47322531
[fractional error] trapz: 1.37961682, simps: 1.37696269, quad: 1.31039310
Number of slices: 64
trapz: -0.48709223, simps: -0.47063369, quad: -0.47322531
[fractional error] trapz: 1.31948854, simps: 1.30869323, quad: 1.31039310
Number of slices: 128
trapz: -0.47636694, simps: -0.47169715, quad: -0.47322531
[fractional error] trapz: 1.31245372, simps: 1.30939076, quad: 1.31039310
Number of slices: 256
trapz: -0.47399035, simps: -0.47296676, quad: -0.47322531
[fractional error] trapz: 1.31089489, simps: 1.31022351, quad: 1.31039310
Number of slices: 512
trapz: -0.47341498, simps: -0.47318938, quad: -0.47322531
[fractional error] trapz: 1.31051750, simps: 1.31036953, quad: 1.31039310
```

[2]:
```python
print('Problem 1: Cosine function with slow variation. All methods give␣
↪accurate results with 16 slices. Simpson\'s and adaptive Gaussian quadrature␣
↪methods are more accurate than the trapezoid method.')
print('')
print('Problem 2: Simple inverse square function. Trapezoid and Simpson\'s␣
↪methods converge quickly with few points. Adaptive Gaussian quadrature␣
↪converges more slowly, around 32 slices are sufficient.')
print('')
print('Problem 3: Quadratic polynomial over [4,2]. Trapezoid and Simpson\'s␣
↪methods perform poorly. Adaptive Gaussian quadrature gives accurate results␣
↪with only 256 slices.')
print('')
print('Problem 4: Cosine function squared. Adaptive Gaussian quadrature␣
↪converges faster than the other methods with 256 slices. Trapezoid and␣
↪Simpson\'s methods oscillate around the exact value.')
print('')
print('Overall, the number of slices needed for accuracy depends on function␣
↪complexity. Some methods are not suitable for certain functions. The␣
↪adaptive Gaussian quadrature is most accurate but expensive for complex␣
↪functions.')
```

```
Problem 1: Cosine function with slow variation. All methods give accurate
results with 16 slices. Simpson's and adaptive Gaussian quadrature methods are
more accurate than the trapezoid method.

Problem 2: Simple inverse square function. Trapezoid and Simpson's methods
converge quickly with few points. Adaptive Gaussian quadrature converges more
slowly, around 32 slices are sufficient.

Problem 3: Quadratic polynomial over [4,2]. Trapezoid and Simpson's methods
```

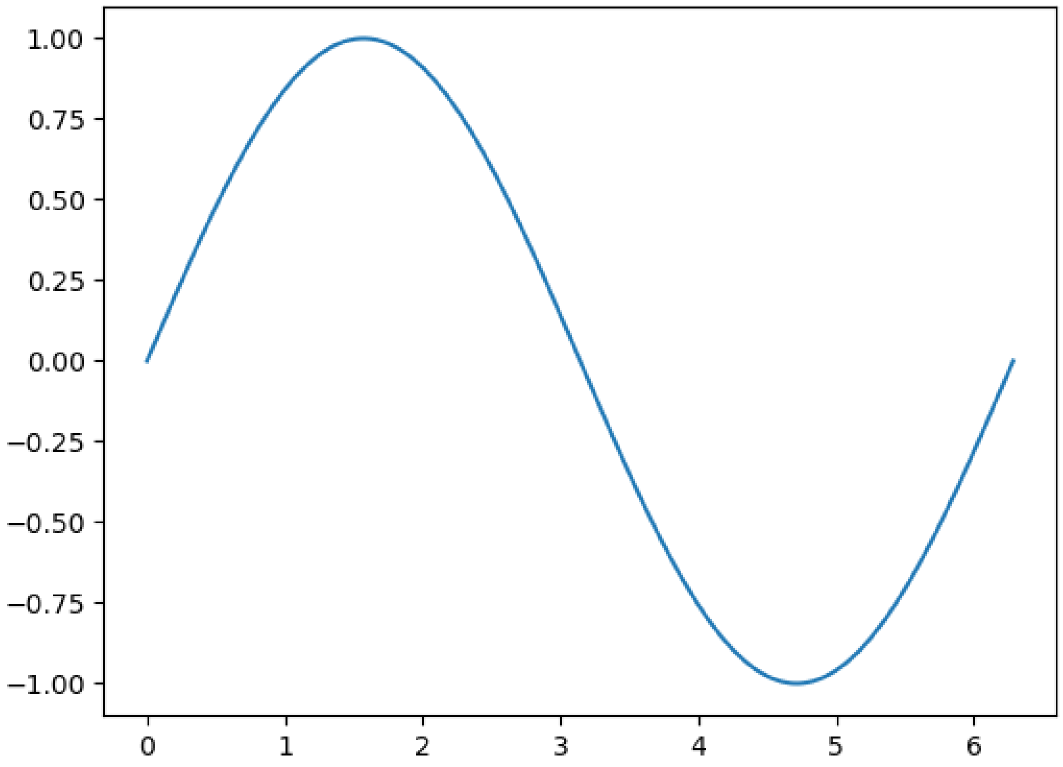
```
perform poorly. Adaptive Gaussian quadrature gives accurate results with only
256 slices.

Problem 4: Cosine function squared. Adaptive Gaussian quadrature converges
faster than the other methods with 256 slices. Trapezoid and Simpson's methods
oscillate around the exact value.

Overall, the number of slices needed for accuracy depends on function
complexity. Some methods are not suitable for certain functions. The adaptive
Gaussian quadrature is most accurate but expensive for complex functions.
```

1.2 Problem 2: Numerical differentiation [Ayars 2.8]

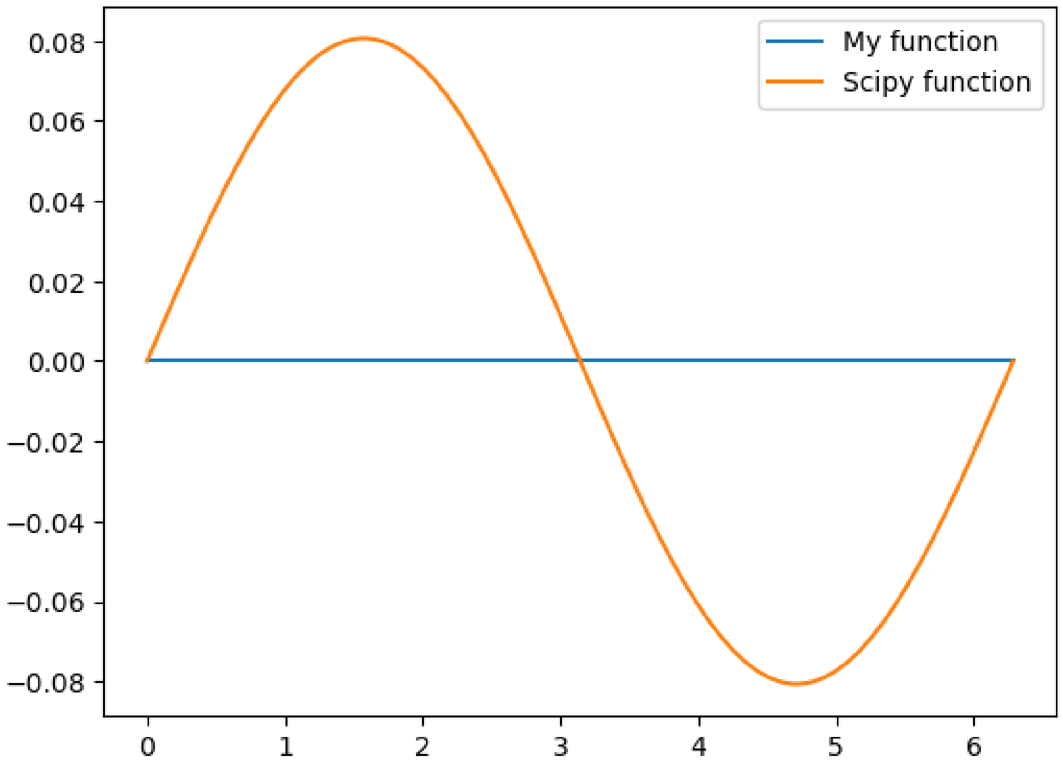
Write a function that, given a list of x-values x_i and function values $f_i(x_i)$, returns a list of values of the second derivative $f''(x_i)$ of the function. Test your function by giving it a list of known function values for $\sin(x)$ and making a graph of the differences between the output of the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

[3]:
```python
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 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'''
    n = len(x_values)
    second_derivatives = np.zeros(n)
    for i in range(1, n - 1):
        h = x_values[i+1] - x_values[i]
        second_derivatives[i] = (function_values[i-1] - 2*function_values[i] +␣
 ↪function_values[i+1]) / h**2
    return second_derivatives


# my function
second_derivatives = second_derivative(x_values, function_values)

# scipy.misc.derivative
second_derivatives_scipy = derivative(np.sin, x_values, n=2)
```

```python
# exact second derivative
exact_second_derivatives = -np.sin(x_values)

plt.plot(x_values, second_derivatives - exact_second_derivatives, label='My␣
 ↪function')
plt.plot(x_values, second_derivatives_scipy - exact_second_derivatives,␣
 ↪label='Scipy function')
plt.legend()
plt.show()
```



```
/tmp/ipykernel_232/2698178517.py:29: DeprecationWarning: scipy.misc.derivative
is deprecated in SciPy v1.10.0; and will be completely removed in SciPy v1.12.0.
You may consider using findiff: https://github.com/maroba/findiff or
numdifftools: https://github.com/pbrod/numdifftools
  second_derivatives_scipy = derivative(np.sin, x_values, n=2)
```



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.

[4]:
```
import numpy as np
import matplotlib.pyplot as plt
plt.rcParams['figure.figsize'] = 10, 5
plt.rcParams['font.size'] = 14
```

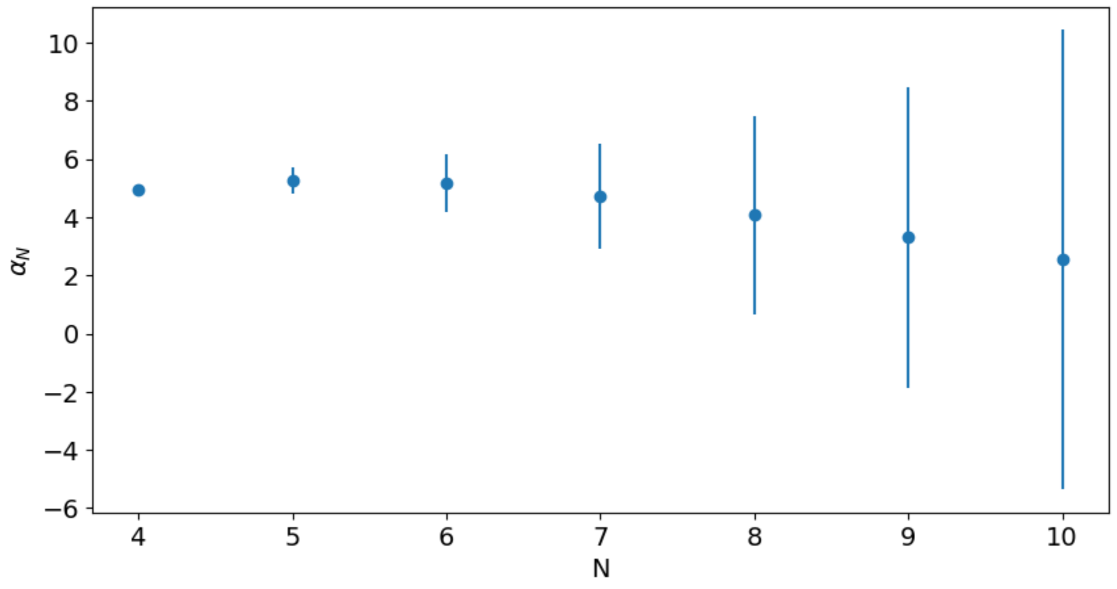
If you're having trouble getting started, we did the $N = 2$ case in Workshop 8. To use this method of MC integration, start by generating a large number (something like 10000) of points randomly scattered in some region of $N-$dimensional space. I say "some region", because you may choose to sample 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).

[5]:
```
# 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
```

[9]:
```
import numpy as np
import matplotlib.pyplot as plt

plt.rcParams['figure.figsize'] = 10, 5
plt.rcParams['font.size'] = 14

def estimate_alpha_N(N, n_trials=1000):
    n_points = 10000

    points = np.random.rand(n_points, N) * 2 - 1

    r2 = np.sum(points**2, axis=1)
    in_sphere = r2 <= 1
    n_in_sphere = np.sum(in_sphere)

    alpha_N = n_in_sphere / n_points * (2 ** N)
    error = np.sqrt(alpha_N * (2 ** N - alpha_N) / n_points)

    for i in range(1, n_trials):
        new_points = np.random.rand(n_points, N) * 2 - 1
        r2 = np.sum(new_points**2, axis=1)
        in_sphere = r2 <= 1
        n_in_sphere = np.sum(in_sphere)
        alpha_N_new = n_in_sphere / n_points * (2 ** N)
        delta = alpha_N_new - alpha_N
        alpha_N += delta / (i + 1)
        error += delta * (alpha_N_new - alpha_N)

    error /= np.sqrt(n_trials)

    return alpha_N, error

N_values = np.arange(4, 11)
alpha_N_values = np.zeros_like(N_values, dtype=float)
```

```
error_values = np.zeros_like(alpha_N_values, dtype=float)

for i, N in enumerate(N_values):
    alpha_N, error = estimate_alpha_N(N)
    alpha_N_values[i] = alpha_N
    error_values[i] = error

plt.errorbar(N_values, alpha_N_values, yerr=error_values, fmt='o')
plt.xlabel('N')
plt.ylabel(r'$\alpha_N$')
plt.show()
```



[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]: