Integration Methods

Numerically computing $\int_0^1 x^2 dx$

```
"""
function that computes square of the given value x
"""
def square(x):
  return x*x
```

Riemann Sum Method

The sum is calculated by partitioning the region into shapes (usually rectangles or trapezoids), calculating area of each shape and adding all the areas. As the number of partitions get arbitrarily large, the Riemann sum gets arbitrarily closer to the actual definite integral.

[Wikipedia Riemann Sum Page]

Left Riemann Sum

The left Riemann Sum is found by approximating the function to its value at the left end point. The width of rectangle may or may not be uniform, but we take uniform partitioning. Once we add up all the areas, we get :

$$A_{left} = \Delta x \cdot [f(a) + f(a + \Delta x) + f(a + 2\Delta x) + \cdots + f(a + (n-1)\Delta x)]$$

```
"""
function to integrate given integrand from 'a' to 'b' using
the left Riemann sum method.

N is the number of rectangles to use
integrand is the function to integrate
a is the lower limit (default is 0)
b is the upper limit (default is 1)
"""
def left_riemann_sum (N, integrand, a = 0.0, b = 1.0) :
  delta_x = (b-a)/N # uniform partition of the interval [a, b]
  sum = 0
  for i in range (N):
    sum = sum + integrand(a + (i * delta_x))
  return sum/N
```

```
left_riemann_sum (10000, square)
```

```
0.3332833349999983
```

Right Riemann Sum

The Right Riemann Sum is found by approximating the funciton to its value at the right end point of the interval. Again, we consider uniform partition of the interval. Once we add up all the areas, we get :

$$A_{right} = \Delta x \cdot [f(a + \Delta x) + f(a + 2\Delta x) + \cdots + f(a + n\Delta x)]$$

```
"""
function to integrate given integrand from 'a' to 'b' using
the right Riemann sum method.

N is the number of rectangles to use
integrand is the function to integrate
a is the lower limit (default is 0)
b is the upper limit (default is 1)
"""
def right_riemann_sum (N, integrand, a = 0.0, b = 1.0) :
  delta_x = (b-a)/N # uniform partition of the interval [a, b]
  sum = 0
  for i in range (N):
    sum = sum + integrand(a + ((i+1) * delta_x))
  return sum/N
```

```
right_riemann_sum(10000, square)
```

```
0.3333833349999983
```

Trapezoidal Riemann Sum

Here, the trapezoidal area is taken, rather than rectangular area. Numerically, it is the average of left and right Riemann sums. When we add all the trapezoidal areas, we get :

$$A_{trpz} = \frac{\Delta x}{2} \cdot [f(a) + 2 \cdot f(a + \Delta x) + 2 \cdot f(a + 2\Delta x) + \cdots 2 \cdot f(a + (n-1)\Delta x) + f(b)]$$

```
"""
function to integrate given integrand from 'a' to 'b' using
the Riemann trapezoidal sum method.

N is the number of rectangles to use
integrand is the function to integrate
a is the lower limit (default is 0)
b is the upper limit (default is 1)
"""
def riemann_trapezoidal_sum(N, integrand, a = 0.0, b = 1.0):
  sum = 0
```

```
  delta_x = (b-a)/N #uniform partition of the interval [a, b]
  for i in range (N):
    left_plus_right = integrand(a + (i * delta_x)) + \
                      integrand(a + ((i+1) * delta_x))
    sum = sum + (left_plus_right/2)
  return sum/N
```

```
riemann_trapezoidal_sum(10000, square)
```

```
0.33333333500000173
```

Simpons method (aka parabolic approximation)

In this method, we will have to partition the interval into an even number of partitions. Unlike the previous methods, here we approximate the area not by straight lines, but by vertical parabolas. Once we calculate the area under each parabola, we will get :

$$\int_a^b f(x) \cdot dx \approx \frac{\Delta x}{3} \cdot (y_0 + 4y_1 + 2y_2 + 4y_3 + \cdots + 4y_{n-1} + y_n)$$

where $\Delta x = \frac{b-a}{n}$ and $y_i = f(a + i\Delta x)$

See also Wikipedia Simpson's_rule

```
"""
integrate using Simpson's method/ parabolic curve approximation,
where N is number of intervals
"""
def simpson_sum(N, integrand, a = 0.0, b = 1.0):
  if N % 2 != 0:
    # we have to make sure that N is even for Simpson's formula.
    # Hence we add 1 if N is odd
    N += 1

  delta_x = (b - a)/N # uniform partition of the interval [a, b]
  sum = 0
  for i in range(0, N-1, 2):
    sum += (delta_x/3) * ( integrand(a + i*delta_x) +
                4*integrand(a + (i+1)*delta_x) + integrand(a + (i+2)*delta_x) )
  return sum
```

```
simpson_sum(1, square)
```

```
0.3333333333333333
```

Monte Carlo Integration Method

The Monte Carlo method uses average value of function values evaluated at randomly chosen discrete points in the integration interval.

Explanation :

- Let the continuous random variable X have a probability density function p(x). Then, the expected value of a function f(X) is :

$$E(f(X)) = \int_{-\infty}^{\infty} f(x) \cdot p(x)dx$$

- If X is a uniform random variable in the interval [a, b] :

$$p(x) = \frac{1}{b-a},$$

and

$$E(f(X)) = \frac{1}{b-a} \cdot \int_{a}^{b} f(x)dx.$$

- Then, $\int_a^b f(x)dx = (b-a) \cdot E(f(X))$. This is the basis for Monte Carlo method.

$$E(f(x)) = \lim_{N\to\infty} \frac{\sum_{i=0}^{N} f(X_i)}{N}$$

$$E(f(x)) \approx \frac{\sum_{i=0}^{N} f(X_i)}{N} \text{ for some large} N$$

$$E(f(X)) \approx \frac{\sum_{i=0}^{N} f(X_i)}{N} \text{ for some large} N$$

$$\int_{a}^{b} f(x)dx \approx (b-a) \cdot \frac{\sum_{i=0}^{N} f(X_i)}{N} \text{ for some large} N$$

- See Also "Basics of Monte Carlo Integration Method"

```
import random

"""
Integrating the integrand by using Monte Carlo method,
where N is the number of points chosen for sampling.
"""
def Monte_Carlo_method(N, integrand, a = 0.0, b = 1.0):
  sum = 0
  for i in range(N):
    sum = sum + integrand(a + random.random()*(b-a))
  return sum/N
```

```
Monte_Carlo_method(10000, square)
```

```
0.3354817310601054
```

Numerically Integrating, given a set of points (x, y) of the function:

- The rectangular Riemann Sum methods now work by simply multiplying y coordinate with the difference of consecutive x coordinates, like : $\Sigma y_i \cdot (x_{i+1} - x_i)$, or $\Sigma y_{i+1} \cdot (x_{i+1} - x_i)$, and the trapezoidal sum works in a similar way too.
- We need an odd number of points (x, y) and even spacing between them (atleast between consecutive points) for Simpson's method to work. As we may not always get such a set of points, Simpson's method is NOT done here.
- For the Monte Carlo method, we average out all the y-coordinates and multiply by the total width, $(x_n - x_1)$.
- The set of points (x, y) is taken as a list of tuples using the python language. The first element of the tuple is the x-coordinate and the second element is the y-coordinate. Each element of a list is a tuple representing (x_i, y_i). We assume that the list is sorted by the first coordinate, i.e $x_1 < x_2 < x_3 < \cdots < x_n$.

```
def left_riemann_sum2(list_of_points):
  N = len(list_of_points)
  sum = 0
  for i in range(N - 1):
    sum = sum + (list_of_points[i][1] *
                 (list_of_points[i+1][0] - list_of_points[i][0]))
  return sum
```

```
left_riemann_sum2([ (i/10000, square(i/10000))  for i in range(10001)])
```

```
0.33328333499999957
```

```
x =  sorted([ random.random() for i in range(10000)])
left_riemann_sum2([(i, square(i)) for i in x])
```

```
0.333147647368984
```

```
def right_riemann_sum2(list_of_points):
  N = len(list_of_points)
  sum = 0
  for i in range(N-1):
    sum = sum + (list_of_points[i+1][1] *
                 (list_of_points[i+1][0] - list_of_points[i][0]))
  return sum
```

```
right_riemann_sum2([ (i/10000, square(i/10000)) for i in range(10001)])
```

```
0.3333833349999992
```

```
x =  sorted([ random.random() for i in range(10000)])
right_riemann_sum2([(i, square(i)) for i in x])
```

```
0.33339650309007884
```

```
def trapezoidal_sum2(list_of_points):
  N = len(list_of_points)
  sum = 0
  for i in range (N-1):
    left_plus_right = (list_of_points[i+1][1] + list_of_points[i][1]) * \
         (list_of_points[i+1][0] - list_of_points[i][0])
    sum = sum + (left_plus_right/2)
  return sum
```

```
trapezoidal_sum2([ ((i/10000), square(i/10000)) for i in range (10001)])
```

```
0.33333333499999973
```

```
def monte_carlo_sum2(list_of_points):
  N = len(list_of_points)
  sum = 0
  for i in range (N):
    sum = sum + list_of_points[i][1]
  integral = (sum/N) * (list_of_points[N-1][0] - list_of_points[0][0])
  return integral
```

```
x =  sorted([ random.random() for i in range(10000)])
monte_carlo_sum2([(i, square(i)) for i in x])
```

```
0.32807657703552123
```

Error analysis :

- The numerically computed integral is an approximation. It will differ from the actual value, and we can achieve arbitrary degree of precision by increasing our sample sizes.
- The absolute error is the absolute value of the difference between the actual value and the value computed by us. Percentage error is relative error, expressed as a percentage.
- Here, the exact value of $\int_0^1 x^2 dx$ is $\frac{1}{3}$ by Newton-Leibniz method.
- We run a python program illustrating the absolute and percentage errors for the different methods used to numerically integrate x^2. The data is expressed in a tabular format to emphasize the increasing accuracy of the computed values.
- For Monte Carlo method, N is the number of sample points, and for all other methods, N is the number of partitions.

```
from tabulate import tabulate # we use the tabulate module to print in a table form

def percent_error(a, b):
  return "{0:.3g}".format(100 * abs(a - b)/a) + "%"

N = [10, 100, 1000, 10000, 100000]
actual = 1/3
# d is a list of lists, and each element in d is a list containing the error inform
d = []
l = []
for i in N:
  computed = left_riemann_sum(i, square)
  l = [i, "left sum", computed, abs(actual - computed),
       percent_error(actual, computed)]
  d.append(l)
for i in N:
  computed = right_riemann_sum(i, square)
  l = [i, "right sum", computed, abs(actual - computed),
       percent_error(actual, computed)]
  d.append(l)
for i in N:
  computed = riemann_trapezoidal_sum(i, square)
  l = [i, "trapezoidal", computed, abs(actual - computed),
       percent_error(actual, computed)]
  d.append(l)
for i in N:
  computed = simpson_sum(i, square)
  l = [i, "Simpson", computed, abs(actual - computed),
       percent_error(actual, computed)]
  d.append(l)
for i in N:
  computed = Monte_Carlo_method(i, square)
  l = [i, "Monte Carlo", computed, abs(actual - computed),
       percent_error(actual, computed)]
  d.append(l)

print(tabulate(d, headers=["Number of samples", "Method",
                           "Computed value", "Error", "% error"]))
```

Number of samples	Method	Computed value	Error	% error
10	left sum	0.285	0.0483333	14.5%
100	left sum	0.32835	0.00498333	1.49%
1000	left sum	0.332833	0.000499833	0.15%
10000	left sum	0.333283	4.99983e-05	0.015%
100000	left sum	0.333328	4.99998e-06	0.0015%
10	right sum	0.385	0.0516667	15.5%
100	right sum	0.33835	0.00501667	1.51%
1000	right sum	0.333833	0.000500167	0.15%
10000	right sum	0.333383	5.00017e-05	0.015%
100000	right sum	0.333338	5.00002e-06	0.0015%
10	trapezoidal	0.335	0.00166667	0.5%
100	trapezoidal	0.33335	1.66667e-05	0.005%
1000	trapezoidal	0.333334	1.66667e-07	5e-05%

10000	trapezoidal	0.333333	1.66667e-09	5e-07%
100000	trapezoidal	0.333333	1.66619e-11	5e-09%
10	Simpson	0.333333	5.55112e-17	1.67e-14%
100	Simpson	0.333333	5.55112e-17	1.67e-14%
1000	Simpson	0.333333	1.11022e-16	3.33e-14%
10000	Simpson	0.333333	1.11022e-16	3.33e-14%
100000	Simpson	0.333333	1.44329e-15	4.33e-13%
10	Monte Carlo	0.173595	0.159738	47.9%
100	Monte Carlo	0.361122	0.0277886	8.34%
1000	Monte Carlo	0.350089	0.016756	5.03%
10000	Monte Carlo	0.334729	0.00139568	0.419%
100000	Monte Carlo	0.333301	3.22797e-05	0.00968%

Observations:

- As the number of samples/partitions increases, the error decreases in all the methods.
- The error in Simpson's rule is exceptionally small, for any amount of partitioning. This is because the Simpson's method is accurate for polynomials of degree 3 or less. Here our integrand is x^2
- The Monte Carlo method usually requires large number of samples as it depends on the "law of large numbers"
- As the number of intervals increases in the left Riemann sum, the computed value increases. This is because x^2 is an increasing function in [0, 1]. Similarly, the right Riemann sum decreases as we increase the number of partitions.

✓ 0s completed at 12:00