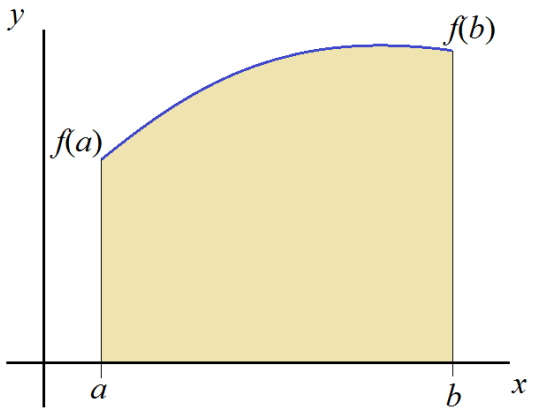
V. Numerical Integration (Quadrature)

The main idea of limited integration is to computed the area under a curve of a function between two limits, a and b, by means of analytical integration rules.



The numerical integration (the quadrature) is to compute the area under the curve of a given function by dividing the area to strips or regions that the area of each of them can be calculated by using simple mathematical rules then all areas are summed to get the total area under the curve from a to b.

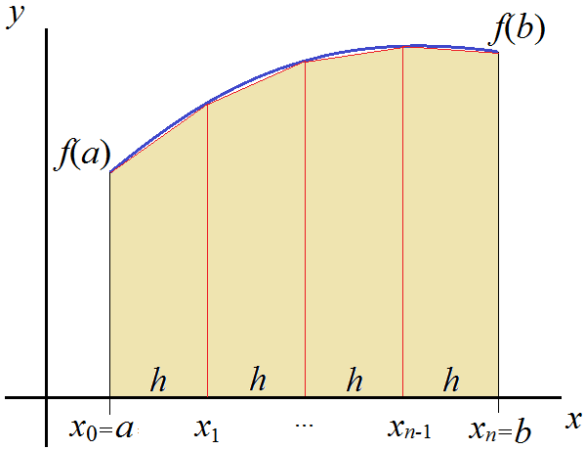
The numerical integration is performed in one (or more) of the following cases:

a) The function does not have an explicit integral
b) Curve of empirical data is to be integrated
c) The integration is a part of a compound numerical method

The accuracy of the numerical integration method highly depends on the number of divisions that cover the area under the curve precisely. The method is more efficient when it yields more accurate result with lesser divisions.

Trapezoidal Rule

The trapezoidal rule is the most basic numerical integration method. As shown in the figure, the area under the curve is divided to vertical trapezoids having equal width, h, and the upper points coincide with the curve.



The area of the first section is

$$A = h[f(x_0) + f(x_1)]/2$$

The integral will be equal to the sum of the trapezoidal areas:

$$I = \frac{h}{2}[f(x_0) + f(x_1)] + \frac{h}{2}[f(x_1) + f(x_2)] + \cdots + \frac{h}{2}[f(x_{n-2}) + f(x_{n-1})] + \frac{h}{2}[f(x_{n-1}) + f(x_n)]$$

So,

$$I = h\left\{\frac{1}{2}[f(x_0) + f(x_n)] + f(x_1) + f(x_2) + \cdots + f(x_{n-2}) + f(x_{n-1})\right\}$$

or

$$I = h\left\{\frac{1}{2}[f(x_a) + f(x_b)] + f(x_1) + f(x_2) + \cdots + f(x_{n-2}) + f(x_{n-1})\right\}$$

Which can be programmed with in a single for-loop. Since the step size, h, is constant, the notation of x_i can be implemented in the code as $x_1 \rightarrow$ `a+h`, $x_2 \rightarrow$ `a+2h` and so on.

Example: Find the value of the integral

$$\int_0^{\frac{\pi}{2}} x \sin x \, dx$$

Solution: The analytical integration gives 1.

The programming can be made in the following steps:

Step 1: Define the function, the limits of the integration and the number of divisions.

```
from math import sin, pi  # standard python math module
def f(x): return x*sin(x) # definition of the function
a = 0                     # the lower limit of the integration
b = pi/2                  # the upper limit of the integration
n = 5                     # the number of divisions
```

Step 2: Calculate the value of step size and initialize summation variable.

```
h = (b-a)/n               # the width of each division
S = 0.5*(f(a)+f(b))       # beginning value of summation
```

Step 3: construct a for-loop from i=1 to n-1 and add values of $f(a + ih)$ to summation variable inside the loop. After the end of the for-loop multiply the value of summation by h to get the value of the integral.

```
for i in range(1,n):      # note: range(1,n) is from 1 to n-1
    S += f(a + i*h)
Integral = h * S
```

Finally, by adding display statements, the code becomes as following:

```
from math import sin, pi
```

```
def f(x): return x*sin(x)
a = 0
b = pi/2
n = 5
h = (b-a)/n
S = 0.5*(f(a)+f(b))
for i in range(1,n):
    S += f(a + i*h)
Integral = h * S
print('Integral = %f' % Integral)
```

The output:

```
Integral = 1.008265
```

In order to reduce the error, let's increase the number of divisions to 10. The result becomes

```
Integral = 1.002059
```

In case of n = 100:

```
Integral = 1.000021
```

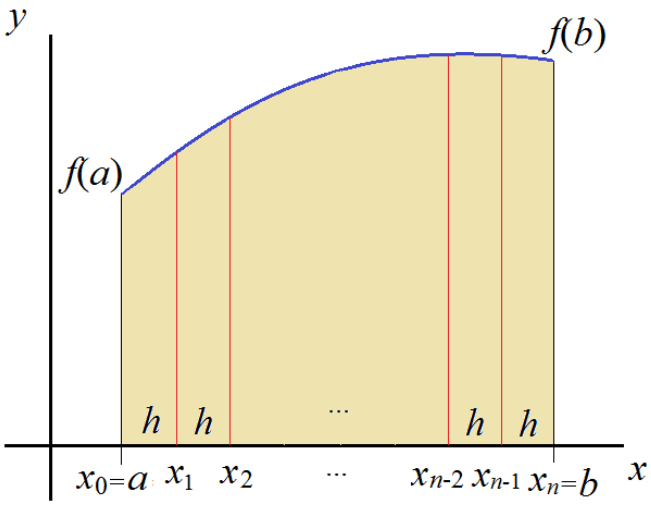
So, using the trapezoidal rule, a 100 divisions are needed to get a solution accurate to the fourth decimal digit in this problem.

Simpson's Rules

Simpson's rules use weighing factors to calculate the integrals to improve the accuracy with lesser number of divisions. Unlike the trapezoidal rule which considers two points, x_i and x_{i+1}, to calculate a trapezoid's area, Simpson's rules use more than two points, i.e. multiple strips, in each turn. The values of $f(x)$ at the multiple points are adjusted by weighing factors to minimize the error.

Simpson's 1/3 rule:

This method calculates the area of two strips at a time, thus, three values of x are taken into account at each turn. To cover the whole domain exactly, the number of strips, n, should even.



The formula of Simpson's 1/3 rule for the first two strips is:

$$A = \frac{1}{3}h[f(x_0) + 4f(x_1) + f(x_2)]$$

Thus, the sum of all strip pairs will be

$$I = \frac{1}{3}h[f(x_0) + 4f(x_1) + f(x_2)] + \frac{1}{3}h[f(x_2) + 4f(x_3) + f(x_4)] + \cdots$$
$$+ \frac{1}{3}h[f(x_{n-4}) + 4f(x_{n-3}) + f(x_{n-2})] + \frac{1}{3}h[f(x_{n-2}) + 4f(x_{n-1}) + f(x_n)]$$

To simplify programming, the formula can be rewritten as

$$I = \frac{1}{3}h\{[f(x_0) + f(x_n)] + 4[f(x_1) + f(x_3) + \cdots + f(x_{n-3}) + f(x_{n-1})]$$
$$+ 2[f(x_2) + f(x_4) + \cdots + f(x_{n-4}) + f(x_{n-2})]\}$$

Finally, putting the terms of similar multipliers in summation form yields

$$I = \frac{1}{3}h\left\{[f(a) + f(b)] + \sum_{i=1,3,5}^{n-1} 4f(x_i) + \sum_{i=2,4,6}^{n-2} 2f(x_i)\right\}$$

With keeping in mind that *n* must be an even number for correct calculation procedure.

The programming procedure is very similar to that of the trapezoidal rule with one additional loop for the second summation. So, the first for-loop will be from 1 to *n*-1 with increment 2

```
S = f(a)+f(b)
for i in range(1,n,2):     # note: range(1,n,2) is from 1 to n-1
    S += 4*f(a + i*h)
```

and the second for-loop will be from 2 to n-2 with increment 2

```
for i in range(2,n,2):     # note: range(2,n,2) is from 2 to n-2
    S += 2*f(a + i*h)
Integral = h/3 * S
```

Example: Find the value of

$$\int_0^{\frac{\pi}{2}} x \sin x \, dx$$

Solution: The analytical integration gives 1.

The program:

```
from math import sin, pi
def f(x): return x*sin(x)
a = 0
b = pi/2
n = 6
h = (b-a)/n
S = f(a)+f(b)
for i in range(1,n,2):
```

```
    S += 4*f(a + i*h)
for i in range(2,n,2):
    S += 2*f(a + i*h)
Integral = h/3 * S
print('Integral = %f'%Integral)
```

The output:

```
Integral = 0.999921
```

If n = 10:

```
Integral = 0.999990
```

At n = 18,

```
Integral = 0.999999
```

This shows that Simpson's 1/3 rule is more efficient than trapezoidal rule.

Simpson's 3/8 rule:

It is very similar to Simpson's 1/3 rule except that three strips are taken into account and, consequently, four points will be included at each time. The formula for the first three strips is

$$A = \frac{3}{8}h[f(x_0) + 3f(x_1) + 3f(x_2) + f(x_3)]$$

Thus, the sum of all strip pairs will be

$$I = \frac{3}{8}h[f(x_0) + 3f(x_1) + 3f(x_2) + f(x_3)] + \frac{3}{8}h[f(x_3) + 3f(x_4) + 3f(x_5) + f(x_6)] + \cdots + \frac{3}{8}h[f(x_{n-3}) + 3f(x_{n-2}) + 3f(x_{n-1}) + f(x_n)]$$

Thus,

$$I = \frac{3}{8}h\left\{[f(a) + f(b)] + \sum_{i=1,4,7}^{n-2} 3[f(x_i) + f(x_{i+1})] + \sum_{i=3,6,9}^{n-3} 2f(x_i)\right\}$$

The number of strips, n, must be a multiple of 3 and summation for-loop increment should be 3. The algorithm can be applied to the previous code by minor changes.

Example: Find the value of

$$\int_0^{\frac{\pi}{2}} x \sin x \, dx$$

Solution: The analytical integration gives 1.

The program:

```
from math import sin, pi
def f(x): return x*sin(x)
a = 0
```

```
b = pi/2
n = 6
h = (b-a)/n
S = f(a)+f(b)
for i in range(1,n,3):    # note: range(1,n,3) is from 1 to n-2
    S += 3*(f(a + i*h) + f(a + (i+1)*h))
for i in range(3,n,3):    # note: range(3,n,2) is from 3 to n-3
    S += 2*f(a + i*h)
Integral = 3*h/8 * S
print('Integral = %f'%Integral)
```

The output:

```
Integral = 0.999819
```

At n = 12:

```
Integral = 0.999989
```

At n = 18:

```
Integral = 0.999998
```

The results show that, for this problem, Simpson's 1/3 rule is more accurate than 3/8 rule for equal or close number or divisions.

The Double Integrations

Double quadrature can be performed by computing the integral by multiple nested loops for each variable of the function from the lower to upper limits.

For Simpson's rules the following modifications should be done:

1- The loops should go through the extended form of the Simpson's integral. For example, for the 1/3 rule the utilized form in the algorithm is

$$I = \frac{1}{3}h[f_0 + 4f_1 + 2f_2 + 4f_3 + 2f_4 + \cdots + 2f_{n-4} + 4f_{n-3} + 2f_{n-2} + 4f_{n-1} + f_n]$$

and for the 3/8 rule it will be

$$I = \frac{3}{8}h[f_0 + 3f_1 + 3f_2 + 2f_3 + 3f_4 + \cdots + 3f_{n-4} + 2f_{n-3} + 3f_{n-2} + 3f_{n-1} + f_n]$$

where $f = f(x, y)$. When double quadrature performed, the factor for each term of the first integration should multiplied by the factor of the term in the second integration.

2- Number of divisions for each quadrature can be different but both should satisfy Simpson's rules condition. For 1/3 rule, it should be an even number and for 3/8 rule it should be a multiple of 3. Accordingly, each integration can have different step size.

3- The final summation should by multiplied by the common factor. For Simpson's 1/3 rule it equals to

$$\frac{h_x}{3}\frac{h_y}{3} = \frac{h_x h_y}{9}$$

and for 3/8 rule it is equal to

$$\frac{3h_x}{8}\frac{3h_y}{8} = \frac{9h_x h_y}{16}$$

Example: Find the value of the integral I by using Simpson's 1/3 double integration.

$$I = \int_{-1}^{1}\int_{1}^{2}(x^2y + xy^2)dxdy$$

Solution: The analytical solution is 1.0

Step 1: Definition of the function, limits of the integration and step sizes.

```
def f(x,y): return x**2*y + x*y**2  # the given function
ax = 1              # lower limit of x
bx = 2              # upper limit of x
ay = -1             # lower limit of y
by = 1              # upper limit of y
nx = 10             # number of divisions in x domain
ny = 10             # number of divisions in y domain
hx = (bx-ax) / nx  # x step size
hy = (by-ay) / ny  # y step size
```

Step 2: Initializing summation variable and construction of summation loops: the outer for *dy* and the inner for *dx* according to the given equation.

```
S = 0                           # initialization of summation
for i in range(0, ny+1):        # loop of the outer integral
    if i==0 or i==ny :
        p = 1                   # factor of the first and last terms
    elif i % 2 == 1 :
        p = 4                   # factor for terms multiplied by 4
    else:
        p = 2                   # factor for terms multiplied by 2
    for j in range(0, nx+1):    # loop of the inner integral
        if j==0 or j==nx :
            q = 1               # factor of the first and last terms
        elif j % 2 == 1 :
            q = 4               # factor for terms multiplied by 4
        else:
            q = 2               # factor for terms multiplied by 2
        S += p*q * f(ax + j*hx, ay + i*hy)
```

Step 3: Multiplication of summation value by the common factor and displaying the result.

```
Integral  =  hx*hy/9 * S
print('Integral = %f' % Integral)
```

The output:

```
Integral = 1.000000
```

Integration in SciPy

Numerical integration functions are imported from `scipy.integrate` module. It contains several quadrature functions for different purposes. In this section, `quad()`, `dblquad()` and `nquad()` are applied to the examples solved above.

```
>>> from scipy.integrate import quad, dblquad, nquad
>>> from numpy import sin, pi
>>> f = lambda x: x*sin(x)
>>> print(quad(f, 0, pi/2))
(1.0, 1.1102230246251565e-14)
```

The function returns a tuple containing the value of integral and estimated absolute error. If the integral value is only required, the statement can be as following

```
>>> I, _ = quad(f, 0, pi/2)
>>> print (I)
1.0
```

```
>>> fn = lambda x,y: x**2*y + x*y**2
>>> ax = 1; bx = 2; ay = -1; by = 1
>>> print(dblquad(fn, ax, bx, lambda y:ay, lambda y:by))
(0.9999999999999999, 4.414734146837848e-14)
```

To explain the sequence of integration limits, variables `ax`, `bx`, `ay` and `by` are used. The limits of the outer integral should be given as constant lambda functions. The function returns a tuple containing the value of integral and estimated absolute error.

```
>>> print(nquad(fn,[[ax, bx], [ay, by]]))
```

(1.0, 4.230171575788777e-14)

The function `nquad()` is used for multiple quadrature. The upper and lower limit of ach integration are given as a sequence of 2 numbers. . The function returns a tuple containing the value of integral and estimated absolute error.

For more information about `scipy.integrate`:
https://docs.scipy.org/doc/scipy/reference/integrate.html