Imports

In [2]:
```
import numpy as np
import matplotlib.pyplot as plt
import timeit
```

Numerical Integration

Introduction

The goal is to solve a definite integral without solving analytically. That is, to solve

$$I = \int_a^b f(x)dx$$

The integral can also be written as:

$$\int_a^b f(x)dx = \lim_{n\to\infty}\sum_{i=1}^{n} f(x_k)\Delta x$$
$$\text{where}$$
$$x_k = a + (b-a)k/n$$
$$\Delta x = (b-a)/n$$

Let's take an example. The function $f(x)$ is defined as:

$$f(x) = 3x^2$$

and the limits are $a = 0$ and $b = 1$.

In [3]:
```
f = lambda x: 3*x**2
F = lambda x: x**3
```

We'll be using another function. This time

$$g(x) = \cos(x)$$

We'll be taking the limit from 0 to 1 (again!). The value of I is:

$$I = \sin(1) - \sin(0) = 0.8414709848078965$$

In [4]:
```
g = lambda x: np.cos(x)
G = lambda x: np.sin(x)
```

In [5]:
```
#error
def error(y_calc, a=0, b=1, int_fn= F):
    y_true = int_fn(b) - int_fn(a)
    absolute_error = np.abs(y_true - y_calc)
    relative_error = absolute_error / y_true
    return absolute_error, relative_error
```

Integration as Sum

In [6]:
```
def integral_as_sum(f, a=0, b=1, n=100, report_error=True, **kwargs):
    h = (b-a)/n
    sum = 0
    for i in range(n):
        sum += f(a+i*h)
    y_true = sum*h
    if report_error:
        absolute_error, relative_error = error(y_true, a, b, **kwargs)
        print("Absolute error:", absolute_error)
        print("Relative error:", relative_error)
    return y_true
```

In [7]:
```
integral_as_sum(f, a=0, b=1, n=100)
```

```
Absolute error: 0.014949999999999908
Relative error: 0.014949999999999908
```

Out[7]: `0.9850500000000001`

In [8]:
```
integral_as_sum(f, a=0, b=1, n=100, int_fn=G)
```

```
Absolute error: 0.1435790151921036
Relative error: 0.1706285989467384
```

Out[8]: `0.9850500000000001`

Of course, this method is very time-consuming as well as not very accurate. What follows, we'll see some more efficient methods to solve this problem.

Newton-Cotes Integration

- Newton-Cotes formulas are the most common numerical integration schemes.
- It replaces a complicated function with an approximating function that is easy to integrate numerically. We write $\int_a^b f(x)dx$ as $\int_a^b f_n(x)dx$ where f_n is a polynomial of degree n:

$$f_n(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n$$

Trapezoidal Rule

The trapezoidal rule is the first order example of the Newton-Cotes closed integration formulas. We approximate the integral as $I = \text{width} \times \text{average height}$. That is:

$$I = (b-a)\frac{f(a)+f(b)}{2}$$

Of course, it won't be very accurate.

In [9]:
```
def simple_trapezoid(f, a, b, report_error=True, **kwargs):
    y_true =  ((b-a)*(f(a)+f(b)))/2
    if report_error:
        absolute_error, relative_error = error(y_true, a, b, **kwargs)
        print("Absolute error:", absolute_error)
        print("Relative error:", relative_error)
    return y_true
simple_trapezoid(f, 0, 1)
```

```
Absolute error: 0.5
Relative error: 0.5
```

Out[9]: `1.5`

In [10]:
```
simple_trapezoid(f, 0, 1, int_fn=G)
```

```
Absolute error: 0.6585290151921035
Relative error: 0.7825926586671819
```

Out[10]: `1.5`

Multiple-Application Trapezoidal Rule

The error in using the single trapezoidal rule in the previous problem was 50%. One way to improve the accuracy of the trapezoidal rule is to divide the integration interval from a to b into a number of segments and apply the method to each segment. For this, we follow as:

$$h = (b-a)/n$$
$$I = h\frac{f(x_0)+f(x_1)}{2} + h\frac{f(x_1)+f(x_2)}{2} + \cdots + h\frac{f(x_n-1)+f(x_n)}{2}$$
$$\text{where}$$
$$x_0 = a$$
$$x_1 = a+h$$
$$\vdots$$
$$x_n = b$$
$$\text{Simplifying the above gives}$$
$$I = h\left[\frac{f(a) + 2\sum_{i=1}^{n-1} f(x_i) + f(b)}{2n}\right]$$

In [11]:
```
def multi_trapzoid(f, a, b, n, report_error=True, **kwargs):
    h = (b-a)/n
    sum = 0
    sum+=f(a)
    sum+=f(b)
    for i in range(1, n):
        sum += 2*f(a+i*h)
    y_true = sum*h/2
    if report_error:
        absolute_error, relative_error = error(y_true, a, b, **kwargs)
        print("Absolute error:", absolute_error)
        print("Relative error:", relative_error)
    return y_true
```

In [12]:
```
multi_trapzoid(f, 0, 1, n=10)
```

```
Absolute error: 0.0050000000000001155
Relative error: 0.0050000000000001155
```

Out[12]: `1.0050000000000001`

In [13]:
```
multi_trapzoid(f, 0, 1, n=10, int_fn=G)
```

```
Absolute error: 0.1635290151921036
Relative error: 0.19433708130701197
```

Out[13]: `1.0050000000000001`

Even with $n = 10$ we get a significant decrease in the error. The error is just about 0.5% for f and 16% for g.

Simpson's Rule

- Another way to improve the accuracy of the trapezoidal rule is to use higher order polynomial in function approximation.
- As compared with Trapezoidal rule (function is approximated by first order polynomial); Simpson's 1/3 rule use second-order Lagrange polynomial for each integrant segment.
- Simpson's 3/8 rule use third-order Lagrange polynomial for each integrant segment. 2nd order (Simpson's 1/3) 3rd order (Simpson's 3/8)

Simpson's 1/3 Rule

Here, the function is approximated by second-order Lagrange polynomial. Using this, the average height of the trapezoid is

$$\frac{f(x_0)+4f(x_1)+f(x_2)}{6}$$

Hence the integral becomes:

$$I = (b-a)\frac{f(x_0)+4f(x_1)+f(x_2)}{6}$$

Here, $x_0 = a, x_2 = b$ and $x_1 = (x_0+x_2)/2$.

In [14]:
```
def simpson13(f, a, b, report_error=True, **kwargs):
    numerator = f(a)+f(b)+ 4*f((a+b)/2)
    denominator = 6
    y_true = (b-a)*numerator/denominator
    if report_error:
        absolute_error, relative_error = error(y_true, a, b, **kwargs)
        print("Absolute error:", absolute_error)
        print("Relative error:", relative_error)
    return y_true
```

In [15]:
```
simpson13(f, 0, 1)
```

```
Absolute error: 0.0
Relative error: 0.0
```

Out[15]: `1.0`

In [16]:
```
simpson13(g, 0, 1, report_error=True, int_fn=G)
```

```
Absolute error: 0.00030110743037536913
Relative error: 0.0003578345965715151
```

Out[16]: `0.8417720922382719`

Great! The error is reduced significantly.

Multiple-Application Simpson's 1/3 Rule

The error in Simpson's 1/3 rule can be decreased by using multiple-application. In this case, one gets:

$$h = (b-a)/n$$
$$\text{The average height of the trapezoid becomes}$$
$$\frac{f(a) + 4\sum_{i=1,3,5}^{n-1} f(x_i) + 2\sum_{i=2,4,6}^{n-2} f(x_i) + f(b)}{3n}$$
$$\text{Hence the integral becomes}$$
$$I = \frac{h}{3}\left[f(a) + 4\sum_{i=1,3,5}^{n-1} f(x_i) + 2\sum_{i=2,4,6}^{n-2} f(x_i) + f(b)\right]$$

Let's implement this method.

In [17]:
```
def simpson13multi(f, a, b, n, report_error=True, **kwargs):
    h = (b-a)/n
    sum = 0
    sum+=f(a)
    sum+=f(b)
    for i in range(1, n, 2):
        sum += 4*f(a+i*h)
    for i in range(2, n, 2):
        sum += 2*f(a+i*h)
    y_true = sum*h/3
    if report_error:
        absolute_error, relative_error = error(y_true, a, b, **kwargs)
        print("Absolute error:", absolute_error)
        print("Relative error:", relative_error)
    return y_true
```

In [18]:
```
simpson13multi(f, 0, 1, n=10)
```

```
Absolute error: 2.220446049250313e-16
Relative error: 2.220446049250313e-16
```

Out[18]: `1.0000000000000002`

In [19]:
```
simpson13multi(g, 0, 1, n=10, int_fn=G)
```

```
Absolute error: 4.6804099407271593e-07
Relative error: 5.562176266595423e-07
```

Out[19]: `0.8414714528488906`

The error is now in orders of 1e-7!

Simpson's 3/8 Rule

In Simpson's 3/8 rule, the function is approximated by third-order Lagrange polynomial. Using this, the average height of the trapezoid is

$$\frac{f(x_0)+3f(x_1)+3f(x_2)+f(x_3)}{8}$$

so, the integral becomes:

$$I = (b-a)\frac{f(x_0)+3f(x_1)+3f(x_2)+f(x_3)}{8}$$

Let's implement this method.

In [20]:
```
def simpson38(f, a, b, report_error=True, **kwargs):
    h = (b-a)/3
    numerator = f(a)+3*f(a+h)+3*f(a+2*h)+f(b)
    denominator = 6
    y_true = (b-a)*numerator/denominator
    if report_error:
        absolute_error, relative_error = error(y_true, a, b, **kwargs)
        print("Absolute error:", absolute_error)
        print("Relative error:", relative_error)
    return y_true
```

In [21]:
```
simpson38(f, 0, 1)
```

```
Absolute error: 0.33333333333333326
Relative error: 0.33333333333333326
```

Out[21]: `1.3333333333333333`

In [22]:
```
simpson38(g, 0, 1, int_fn=G)
```

```
Absolute error: 0.28066816971596964
Relative error: 0.33354467923816145
```

Out[22]: `1.1221391545238661`

Multiple-Application Simpson's 3/8 Rule

To use multiple Simpson's 3/8 rule, we need to implement the following:

$$h = (b-a)/n$$

The average height of the trapezoid is

$$3\frac{f(a) + 3\sum_{i\neq 3k}^{n-1} f(x_i) + 2\sum_{i=1}^{n/3-1} f(x_{3i}) + f(b)}{8n}$$

hence the integral becomes:

$$I = \frac{3h}{8}\left[f(a) + 3\sum_{i\neq 3k}^{n-1} f(x_i) + 2\sum_{i=1}^{n/3-1} f(x_{3i}) + f(b)\right]$$

In [23]:
```
def simpson38multi(f, a, b, n, report_error=True, **kwargs):
    h = (b-a)/n
    sum = 0
    sum+=f(a)
    sum+=f(b)
    for i in range(1, n):
        if n%3 == 0:
            sum += 2*f(a+i*h)
            continue
        sum += 3*f(a+i*h)
    y_true = 3*h*sum/8
    if report_error:
        absolute_error, relative_error = error(y_true, a, b, **kwargs)
        print("Absolute error:", absolute_error)
        print("Relative error:", relative_error)
    return y_true
```

In [24]:
```
simpson38multi(f, 0, 1, n=10)
```

```
Absolute error: 0.0743750000000003
Relative error: 0.0743750000000003
```

Out[24]: `1.0743750000000003`

In [25]:
```
simpson38multi(g, 0, 1, n=10, int_fn=G)
```

```
Absolute error: 0.07551419430654838
Relative error: 0.08974069893068015
```

Out[25]: `0.9169851791144449`

Gauss Quadrature

In [ ]:

Final Thoughts

Simpson's 1/3 rule seems to perform the best. Let's compare the time taken by these functions

In [31]:
```
time1 = timeit.timeit(stmt="integral_as_sum(f, a=0, b=1, n=100, report_error=False)",
time2 = timeit.timeit(stmt="simple_trapezoid(f, 0, 1, report_error=False)", number=20
time3 = timeit.timeit(stmt="multi_trapzoid(f, 0, 1, n=100, report_error=False)", numbe
time4 = timeit.timeit(stmt="simpson13(f, 0, 1, report_error=False)", number=2000, setu
time5 = timeit.timeit(stmt="simpson38(f, 0, 1, report_error=False)", number=2000, setu
time6 = timeit.timeit(stmt="simpson38multi(f, 0, 1, n=100, report_error=False)", numbe
time7 = timeit.timeit(stmt="simpson13multi(f, 0, 1, n=100, report_error=False)", numbe
```

In [32]:
```
time1, time2, time3, time4, time5, time6, time7
```

Out[32]:
```
(0.24600010000006023,
 0.005095399999959227,
 0.1765287000000626,
 0.005756799999971918,
 0.007628600000089136,
 0.20141980000005333,
 0.17621189999999842)
```

Simpson's 1/3 rule is faster than Simpson's 3/8 rule. Both when using a single integration or using multiple integration.

Let's compare the errors.

In [43]:
```
f = lambda x: 3*x**2
F = lambda x: x**3
print("Integral as Sum")
integral_as_sum(f, a=0, b=1, n=100, report_error=True)
print("Simple Trapezoid")
simple_trapezoid(f, 0, 1, report_error=True)
print("Multi Trapezoid")
multi_trapzoid(f, 0, 1, n=100, report_error=True)
print("Simpson 1/3")
simpson13(f, 0, 1, report_error=True)
print("Simpson 1/3 Multi")
simpson13multi(f, 0, 1, n=100, report_error=True)
print("Simpson 3/8")
simpson38(f, 0, 1, report_error=True)
print("Simpson 3/8 Multi")
simpson38multi(f, 0, 1, n=100, report_error=True)
```

```
Integral as Sum
Absolute error: 0.014949999999999908
Relative error: 0.014949999999999908
Simple Trapezoid
Absolute error: 0.5
Relative error: 0.5
Multi Trapezoid
Absolute error: 5.0000000000105516e-05
Relative error: 5.0000000000105516e-05
Simpson 1/3
Absolute error: 0.0
Relative error: 0.0
Simpson 1/3 Multi
Absolute error: 0.0
Relative error: 0.0
Simpson 3/8
Absolute error: 0.33333333333333326
Relative error: 0.33333333333333326
Simpson 3/8 Multi
Absolute error: 0.11943124999999966
Relative error: 0.11943124999999966
```

Out[43]: `1.1194312499999997`

In [44]:
```
g = lambda x: np.cos(x)
G = lambda x: np.sin(x)
print("Integral as Sum")
integral_as_sum(g, a=0, b=1, n=10, report_error=True, int_fn=G)
print("Simple Trapezoid")
simple_trapezoid(g, 0, 1, report_error=True, int_fn=G)
print("Multi Trapezoid")
multi_trapzoid(g, 0, 1, n=10, report_error=True, int_fn=G)
print("Simpson 1/3")
simpson13(g, 0, 1, report_error=True, int_fn=G)
print("Simpson 1/3 Multi")
simpson13multi(g, 0, 1, n=10, report_error=True, int_fn=G)
print("Simpson 3/8")
simpson38(g, 0, 1, report_error=True, int_fn=G)
print("Simpson 3/8 Multi")
simpson38multi(g, 0, 1, n=10, report_error=True, int_fn=G)
```

```
Integral as Sum
Absolute error: 0.02228354198711624
Relative error: 0.02648165223689021
Simple Trapezoid
Absolute error: 0.07131983187382662
Relative error: 0.08475613914377401
Multi Trapezoid
Absolute error: 0.0007013427194768607
Relative error: 0.000833472255299419
Simpson 1/3
Absolute error: 0.00030110743037536913
Relative error: 0.0003578345965715151
Simpson 1/3 Multi
Absolute error: 4.6804099407271593e-07
Relative error: 5.562176266595423e-07
Simpson 3/8
Absolute error: 0.28066816971596964
Relative error: 0.33354467923816145
Simpson 3/8 Multi
Absolute error: 0.07551419430654838
Relative error: 0.08974069893068015
```

Out[44]: `0.9169851791144449`

Simpson's 1/3 rule with multiple integration has the lowest error rate, followed by multiple trapezoidal rule. Let's try a somewhat complicated function.

$$f(x) = e^x\sin(x)$$

The definite integral is:

$$I = e^x(\sin(x) - \cos(x)) + c$$

In [45]:
```
f = lambda x: np.exp(x)*np.sin(x)
F = lambda x: np.exp(x)*(np.sin(x)-np.cos(x))
print("Integral as Sum")
integral_as_sum(f, a=0, b=1, n=1000, report_error=True)
print("Simple Trapezoid")
simple_trapezoid(f, 0, 1, report_error=True)
print("Multi Trapezoid")
multi_trapzoid(f, 0, 1, n=1000, report_error=True)
print("Simpson 1/3")
simpson13(f, 0, 1, report_error=True)
print("Simpson 1/3 Multi")
simpson13multi(f, 0, 1, n=1000, report_error=True)
print("Simpson 3/8")
simpson38(f, 0, 1, report_error=True)
print("Simpson 3/8 Multi")
simpson38multi(f, 0, 1, n=1000, report_error=True)
```

```
Integral as Sum
Absolute error: 0.09181277434133606
Relative error: 0.09181277434133606
Simple Trapezoid
Absolute error: 0.14367764358942114
Relative error: 0.14367764358942114
Multi Trapezoid
Absolute error: 0.0906690966977467
Relative error: 0.0906690966977467
Simpson 1/3
Absolute error: 0.09181472999444973
Relative error: 0.09181472999444973
Simpson 1/3 Multi
Absolute error: 0.09066932636854286
Relative error: 0.09066932636854286
Simpson 3/8
Absolute error: 0.21175433103156815
Relative error: 0.21175433103156815
Simpson 3/8 Multi
Absolute error: 0.022568387098687248
Relative error: 0.022568387098687248
```

Out[45]: `1.0225683870986872`

This time Simpson's 3/8 rule performs the best.