In [2]:

```python
#Q(2)(a)

#Import the necessary library
import numpy as np

#Define the function to be integrated from the statement
def f(x):
    return 1/((1+x)**2)

#Define the trapezoid rule function
def trapezoid_rule(a, b, n, func):
    h = (b - a) / n #Calculate the step size
    x = np.linspace(a, b, n + 1) #Generate equally spaced points from a to b
    y = func(x) #Evaluate the function at these points
    return h * (np.sum(y) - 1/2 * (y[0] + y[-1])) #Calculate the trapezoidal sum

#Define the limits of integration
a = 0
b = 2

exact_integral = 2/3  #The exact integral of 1/((1+x)^2) from 0 to 2 is 2/3

#Initialize the variable to store the previous error
error_prev = None

#Test the trapezoid rule function with different numbers of subdivisions
for N in [20, 40, 80]:
    approx_integral = trapezoid_rule(a, b, N, f) #Calculate the approximate integral using the trapezoid rule
    error = abs(exact_integral - approx_integral) #Calculate the absolute error between the exact and approximate integral
    print(f"If h = {2/N}, the error is {error}") #Print the step size and the corresponding error

    #If a previous error exists, calculate and print the ratio of successive errors
    if error_prev is not None:
        ratio = error_prev / error
        print(f"The ratio of successive errors is {ratio}")

    error_prev = error #Update the previous error
```

```
If h = 0.1, the error is 0.001601642128346903
If h = 0.05, the error is 0.00040102746246584164
The ratio of successive errors is 3.9938465024283123
If h = 0.025, the error is 0.00010029568053082638
The ratio of successive errors is 3.9984519806172893
```

In [5]:

```python
##From the output above, we can observe that as the step size h decreases by a factor of 2 (from 0.1 to 0.05, and from 0.05 to 0.025),
#the error approximately decreases by a factor of 4. This is a characteristic of second order convergence.
```

In [7]:

```python
#Q(2)(b)

#Define the function to be integrated from the statement
def f(x):
    return np.sqrt(x)

#Define the limits of integration
a = 0
b = 1

exact_integral = 2/3  #The exact integral of sqrt(x) from 0 to 1 is 2/3

#Initialize the variable to store the previous error
error_prev = None

#Test the trapezoid rule function with different numbers of subdivisions
for N in [16, 32, 64, 128]:
    approx_integral = trapezoid_rule(a, b, N, f) #Calculate the approximate integral using the trapezoid rule
    error = abs(exact_integral - approx_integral) #Calculate the absolute error between the exact and approximate integral
    print(f"If h = {N}, the error is {error}") #Print the step size and the corresponding error

    #If a previous error exists, calculate and print the ratio of successive errors
    if error_prev is not None:
        ratio = error_prev / error
        print(f"The ratio of successive errors is {ratio}")

    error_prev = error #Update the previous error
```

```
If h = 16, the error is 0.0030854697894384664
If h = 32, the error is 0.0011077303877248257
The ratio of successive errors is 2.785397804041227
If h = 64, the error is 0.0003958552881596633
The ratio of successive errors is 2.798321560575026
If h = 128, the error is 0.00014100936984062784
The ratio of successive errors is 2.8072977604755516
```

In [8]:

```python
##From the output above, we can observe that as the number of intervals N doubles (from 16 to 32, 32 to 64, and 64 to 128),
#the error approximately reduces to a quarter of its previous value. This is an indication of second order convergence.
#While these ratios are not exactly 4, they do indicate that the error is decreasing roughly in line with the square
#of the increase in N, which suggests second order convergence.
```

In [11]:

```python
#Q(3)(a)

#Define the function to be integrated from the statement
def f(x):
    return np.cos(x**2)

#Define the trapezoid rule function
def trapezoid_rule(a, b, n, func):
    h = (b - a) / n #Calculate the step size
    x = np.linspace(a, b, n + 1) #Generate equally spaced points from a to b
    y = func(x) #Evaluate the function at these points
    return h * (np.sum(y) - 1/2 * (y[0] + y[-1]))    #Calculate the trapezoidal sum

#Define the function to find a suitable h and calculate the error
def find_h_and_error(func, a, b, target_q, h_start):
    h = h_initial #Start with the initial h
    #Keep looping until the desired accuracy is achieved
    while True:
        #Calculate the trapezoidal sum for h, h/2, and h/4
        T_h = trapezoid_rule(a, b, int((b - a) / h), func)
        T_h_half = trapezoid_rule(a, b, int((b - a) / (h / 2)), func)
        T_h_quarter = trapezoid_rule(a, b, int((b - a) / (h / 4)), func)

        #Calculate q(h)
        q_h = (T_h_half - T_h) / (T_h_quarter - T_h_half)
        #If q(h) is close enough to the target value, break the loop
        if np.isclose(q_h, target_q, rtol=1e-2):
            break

        h /= 2  #Otherwise, halve h for the next iteration

    #Calculate the error
    error = abs(T_h_half - T_h)
    return h, error, T_h, T_h_half

#Define the limits of integration
a = 0
b = np.sqrt(np.pi / 2)

#Define the initial h and the target q
h_initial = 0.1
target_q = 4

#Call the function to find h and calculate the error
h, error, T_h, T_h_half = find_h_and_error(f, a, b, target_q, h_initial)

#Print the result
print(f"The value of h for which q(h) is approximately 4 is {h}")
```

```
The value of h for which q(h) is approximately 4 is 0.05
```

In [12]:

```python
#Q(3)(b)

#Print the result by using output and function in (a)
print(f"The approximation of the error is {error}")
```

```
The approximation of the error is 0.00039386975721300566
```

In [13]:

```python
#Q(3)(c)

#Calculate the improved approximation
S_h = T_h + 4/3 * (T_h_half - T_h)

#Print the result
print(f"The improved approximation is {S_h}")
```

```
The improved approximation is 0.9774514588255858
```

In [15]:

```python
#Q(3)(d)

#The reason why S_h[cos(x**2)] is more accurate and convergent faster is that S_h[cos(x**2)] includes an error correction term.
#And, by adding By adding this error correction term to T_h[cos(x**2)], we effectively reduce the error in the approximation.
```

In [5]:

```python
#Q(4)

#Define the function to be integrated from the statement
def f(x):
    return 1 / (1 + np.sin(x)**2)

#Define the trapezoid rule function
def trapezoid_rule(a, b, n, func):
    h = (b - a) / n #Calculate the step size
    x = np.linspace(a, b, n + 1) #Generate equally spaced points from a to b
    y = func(x) #Evaluate the function at these points
    return h * (np.sum(y) - 1/2 * (y[0] + y[-1])) #Calculate the trapezoidal sum

#Define the limits of integration
a = 0
b = 2*np.pi

#Define the exact value of the integral
exact_value_integral = np.sqrt(2*np.pi)

#Test the trapezoid rule function with different numbers of subdivisions
for N in [10, 20, 40, 80, 160, 320]:
    h = (b - a) / N #Calculate the step size
    approx_integral = trapezoid_rule(a, b, N, f) #Calculate the approximate integral using the trapezoid rule
    error = abs(exact_value_integral - approx_integral) #Calculate the absolute error between the exact and approximate integral

    #print the result
    print(f"If h = {h:.10f}, T_h = {approx_integral:.10f}, Error = {error:.10f}")
```

```
For h = 0.6283185307, T_h = 4.4442042417, Error = 1.9375759670
For h = 0.3141592654, T_h = 4.4428831346, Error = 1.9362548599
For h = 0.1570796327, T_h = 4.4428829382, Error = 1.9362546635
For h = 0.0785398163, T_h = 4.4428829382, Error = 1.9362546635
For h = 0.0392699082, T_h = 4.4428829382, Error = 1.9362546635
For h = 0.0196349541, T_h = 4.4428829382, Error = 1.9362546635
```

In [ ]:

```python
##From h=0.6283185307 to h = 0.1570796327, we can see that as h decreaes, the error decreases too.This suggests that the approximation is converging to the exact solution,
#which is what we expect for a numerical integration method like the trapezoidal rule.
#However, after h=0.1570796327, we can see that as h decreaes, the error holds at Error = 1.9362546635, which suggests that the method has reached
#its limit of precision for this particular function and interval
```