Ramakrishna Mission Vivekananda Centenary College
Rahara, Kolkata, West Bengal - 700118
College with Potential for Excellence (CPE)
Accredited by NAAC with Grade A

PHSA CC-V: Mathematical Physics II

Laboratory Notebook

Rohan Deb Sarkar
Undergraduate Student
Department of Physics
Ramakrishna Mission Vivekananda Centenary College

End Semester Examination

Subject Code:	**PHSA CC-V**
Registration Number:	**A01-1112-111-020-2021**
Eamination Roll Number:	**...**

December, 2022

Contents

1 Matrix Multilpication **2**
1.1 Code . . . 2
1.2 Examples . . . 2

2 Trace of a Matrix **4**
2.1 Code . . . 4
2.2 Examples . . . 4
2.3 Trace of AB = Trace of BA . . . 5

3 Trapezium method for integration **6**
3.1 Code . . . 6
3.2 Examples . . . 7

4 Simpson's method for integration **12**
4.1 Code . . . 12
4.2 Examples . . . 13

5 Bisection method for finding root **18**
5.1 Code . . . 18
5.2 Examples . . . 19

6 Newton-Raphson method for finding root **20**
6.1 Code . . . 20
6.2 Examples . . . 21

7 Fourier series **22**
7.1 Code . . . 22
7.2 Examples . . . 24

8 Fourier transformation **28**
8.1 Code . . . 28
8.2 Examples . . . 29

9 Euler method for solving first-order differential equations **35**
9.1 Code . . . 35
9.2 Examples . . . 36

10 Euler method for solving second-order differential equations **39**
10.1 Code . . . 39
10.2 Examples . . . 40
10.3 Real world physics problems . . . 42

1 Matrix Multilpication

If A is a $m \times n$ matrix and B is a $n \times p$ matrix. Then, matrix product $C_{m\times p} = A_{m\times n} \cdot B_{n\times p}$ is defined as,

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

where, a_{ij}, b_{ij}, c_{ij} represent individual elements of the matrices A, B, and C respectively.

1.1 Code

[1]:
```
from numpy import array, zeros, random
```

[2]:
```
# Code for performing matrix multiplication
def matrix_multiply(A, B):

    # Checking if matrix multiplication is possible
    if not A.shape[1] == B.shape[0]:
        raise ValueError("Matrix multiplication not possible!")

    # Decalring an empty matrix to store the result
    C = zeros((A.shape[0], B.shape[1]))

    # Performing matrix multiplication
    for i in range(A.shape[0]):
        for j in range(B.shape[1]):
            for k in range(A.shape[1]):
                C[i][j] += (A[i][k] * B[k][j])

    return C
```

1.2 Examples

1.2.1

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \times \begin{bmatrix} 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{bmatrix}$$

[3]:
```
# Declaring two matrices `A` and `B`
A = array([[1, 2, 3],
           [4, 5, 6],
           [7, 8, 9]])

B = array([[9, 8, 7],
           [6, 5, 4],
           [3, 2, 1]])

# Performing matrix multipliaction
matrix_multiply(A, B)
```

[3]:
```
array([[ 30.,  24.,  18.],
       [ 84.,  69.,  54.],
       [138., 114.,  90.]])
```

1.2.2

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 9 & 8 \\ 6 & 5 \\ 3 & 2 \end{bmatrix}$$

[4]:
```
# Declaring two non square matrices `A` and `B`
A = array([[1, 2, 3],
           [4, 5, 6]])

B = array([[9, 8],
           [6, 5],
           [3, 2]])

# Performing matrix multipliaction between two non square matrices
matrix_multiply(A, B)
```

[4]:
```
array([[30., 24.],
       [84., 69.]])
```

1.2.3 Matrix multiplication using two randomly generated matrices

[5]:
```
# Generating two random matrices `A` and `B`
A = random.randint(1, 100, (3, 3))
B = random.randint(1, 100, (3, 3))
```

[6]:
```
A
```

[6]:
```
array([[91, 99, 98],
       [50, 82, 56],
       [98, 77, 92]])
```

[7]:
```
B
```

[7]:
```
array([[95, 30, 68],
       [87, 37, 67],
       [37, 72, 72]])
```

[8]:
```
# Performing matrix multipliaction between two random matrices
matrix_multiply(A, B)
```

[8]:
```
array([[20884., 13449., 19877.],
       [13956.,  8566., 12926.],
       [19413., 12413., 18447.]])
```

2 Trace of a Matrix

If A is a $n \times n$ square matrix. Then, trace of matrix $A_{n\times n}$ is defined as,

$$tr(A) = \sum_{i=1}^{n} a_{ii} = a_{11} + a_{22} + a_{33} + \cdots + a_{nn}$$

where, a_{ij} represent individual elements of the matrices A, B, and C respectively.

2.1 Code

[1]:
```
from numpy import array, random, zeros
```

[2]:
```
# Code for calculating trace of a matrix
def matrix_trace(A):

    # Checking if the matrix is a square matrix
    if not A.shape[0] == A.shape[1]:
        raise ValueError(
            "Trace of a matrix can only be calculated for a square matrix!")

    # Intialising variable `trace` to store the trace of the matrix
    trace = 0

    # Calculating the trace of the matrix
    for i in range(A.shape[0]):
        trace += A[i][i]

    return trace
```

2.2 Examples

2.2.1

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

[3]:
```
# Declaring a matrix `A`
A = array([[1, 2, 3],
           [4, 5, 6],
           [7, 8, 9]])

# Calculating the trace of the matrix
matrix_trace(A)
```

[3]:
```
15
```

2.2.2 Trace of a randomly generated matrix

[4]:
```
# Generating a random matrix `A`
A = random.randint(1, 100, (3, 3))
```

[5]:
```
A
```

[5]:
```
array([[93, 34, 23],
       [29, 36, 75],
       [72,  5, 21]])
```

[6]:
```python
# Calculating the trace of the randomly generated matrix
matrix_trace(A)
```

[6]:
```
150
```

2.3 Trace of AB = Trace of BA

For any two square matrices A and B, $tr(AB) = tr(BA)$

[8]:
```python
# Importing the code for matrix multiplication
def matrix_multiply(A, B):

    # Checking if matrix multiplication is possible
    if not A.shape[1] == B.shape[0]:
        raise ValueError("Matrix multiplication not possible!")

    # Decalring an empty matrix to store the result
    C = zeros((A.shape[0], B.shape[1]))

    # Performing matrix multiplication
    for i in range(A.shape[0]):
        for j in range(B.shape[1]):
            for k in range(A.shape[1]):
                C[i][j] += (A[i][k] * B[k][j])

    return C
```

2.3.1 Verifying $tr(AB) = tr(BA)$ using two randomly generated matrices

[9]:
```python
# Generating two random matrices `A` and `B`
A = random.randint(1, 10, (3, 3))
B = random.randint(1, 10, (3, 3))
```

[10]:
```python
A
```

[10]:
```
array([[3, 6, 7],
       [9, 2, 4],
       [1, 6, 7]])
```

[11]:
```python
B
```

[11]:
```
array([[4, 4, 7],
       [7, 8, 1],
       [9, 8, 3]])
```

[12]:
```python
matrix_trace(matrix_multiply(A, B))
```

[12]:
```
235.0
```

[13]:
```python
matrix_trace(matrix_multiply(B, A))
```

[13]:
```
235.0
```

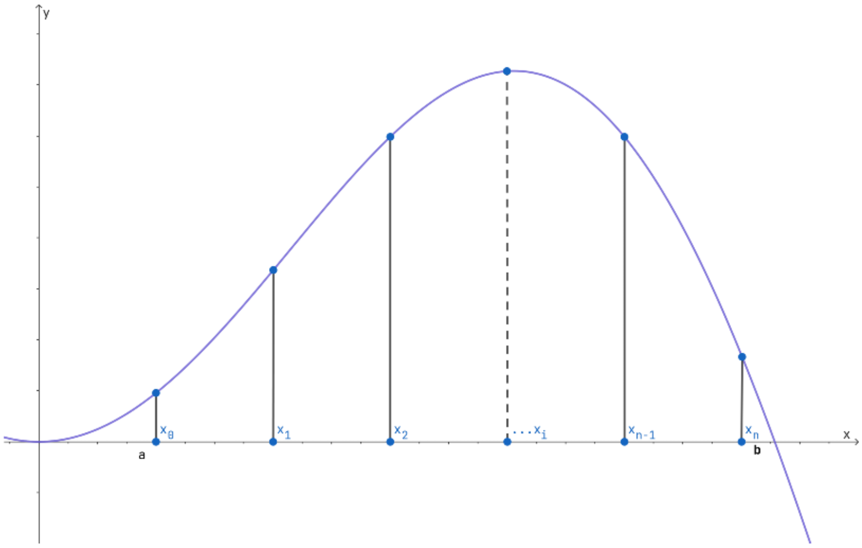
3 Trapezium method for integration

In calculus, the trapezium rule is a method for approximating definite integrals. The trapezium rule works by approximating region under the curve of a function $f(x)$ as a trapezoid (approximating the function using straight line approximations).

Using just one trapezoid the approximation becomes,

$$\int_a^b f(x)dx \approx (b-a) \cdot \frac{1}{2} (f(b) + f(a))$$

When approximating using n points (i.e., $n-1$ trapezoids), we would have to evaluate the function $f(x)$ at those n points,



and then the approximation would be,

$$\int_a^b f(x)dx \approx \frac{h}{2} (f(x_0) + 2f(x_1) + 2f(x_2) + \cdots + 2f(x_{n-1}) + f(x_n))$$

$$\implies \int_a^b f(x)dx \approx \frac{h}{2} \left(f(x_0) + 2\sum_{i=1}^{n-1} f(x_i) + f(x_n) \right)$$

It is quite exident from the above illustration that as n increases the value of integration will be more and more accurate. Therefore as $n \to \infty$, our approximation will reach its exact value.

3.1 Code

[1]:
```python
from numpy import empty, pi, sin, exp
from matplotlib import pyplot as plt

# Default configuaration for matplotlib
plt.style.use(['science', 'ieee'])
plt.rcParams["figure.figsize"] = (10, 5)
```

[2]:
```python
# Function to generate `n` evenly spaced points between limits `a` and `b`
def generate_points(a, b, n, retstep=False):

    # Calculating the spacing (difference) between each points
    h = (b - a) / (n - 1)
```

```
    # Creating an empty array to store the points
    points = empty(n)

    # Generating the points
    for i in range(n):
        points[i] = a
        a += h

    # Returning the difference between each points if asked for
    if retstep:
        return points, h

    return points
```

[3]:
```
def integrate_trapezium(f, x_i, x_f, n):

    # Generating points
    x, h = generate_points(x_i, x_f, n, retstep=True)

    # Evaluating the function `f(x)` for each points
    y = f(x)

    # Declaring a variable to store the integral
    integral = 0

    # Evaluating the integral using Trapezium method
    for i in range(1, n):
        integral += y[i]

    integral = (h / 2) * (y[0] + 2 * integral + y[n - 1])

    return integral
```

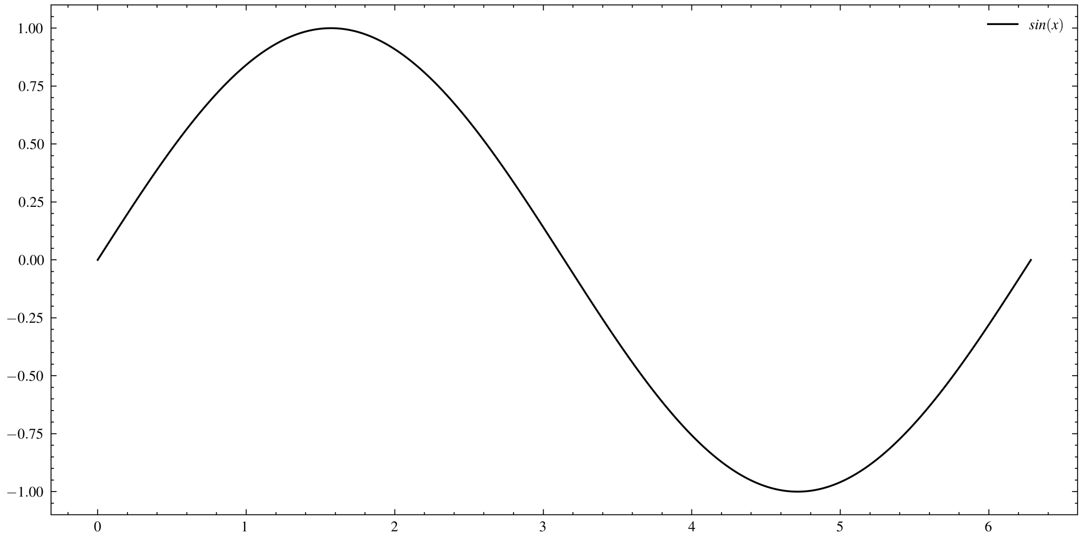
3.2 Examples

3.2.1

$$\int_0^{2\pi} sin(x) \cdot dx$$

[4]:
```
def f(x):
    return sin(x)


x_i = 0
x_f = 2 * pi
n = 1000
```

[5]:
```
x = generate_points(x_i, x_f, n)
plt.plot(x, f(x), label="$sin(x)$")
plt.legend()
plt.show()
```



[6]:
```
integrate_trapezium(f, x_i, x_f, n)
```

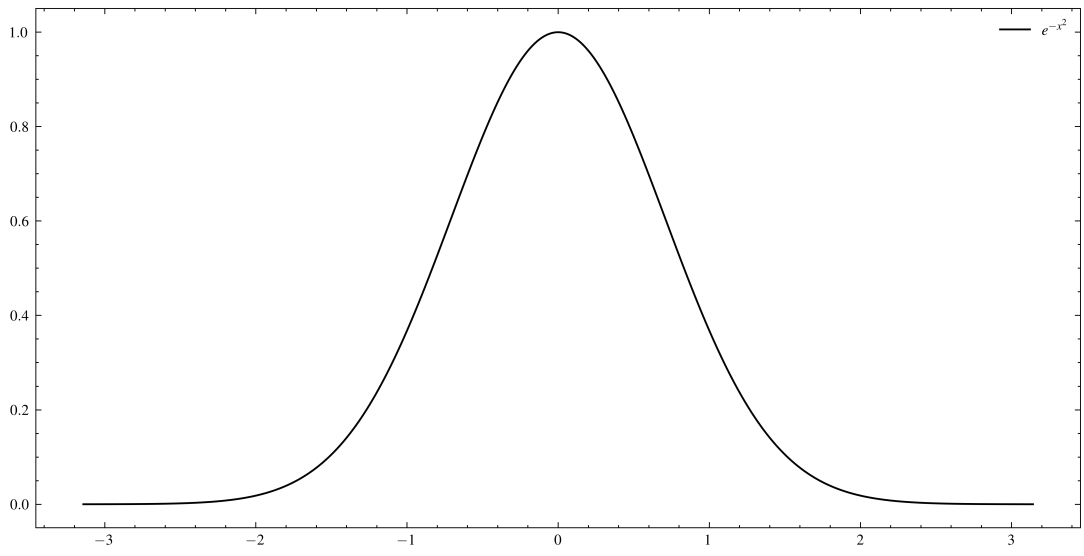
[6]: 1.1556633208689454e-13

3.2.2

$$\int_{-\pi}^{\pi} e^{-x^2} \cdot dx$$

[7]:
```
def f(x):
    return exp(-(x**2))


x_i = - pi
x_f = pi
n = 1000
```

[8]:
```
x = generate_points(x_i, x_f, n)
plt.plot(x, f(x), label="$e^{-x^2}$")
plt.legend()
plt.show()
```



[9]:
```
integrate_trapezium(f, x_i, x_f, n)
```

[9]: 1.7724384415148215

3.2.3

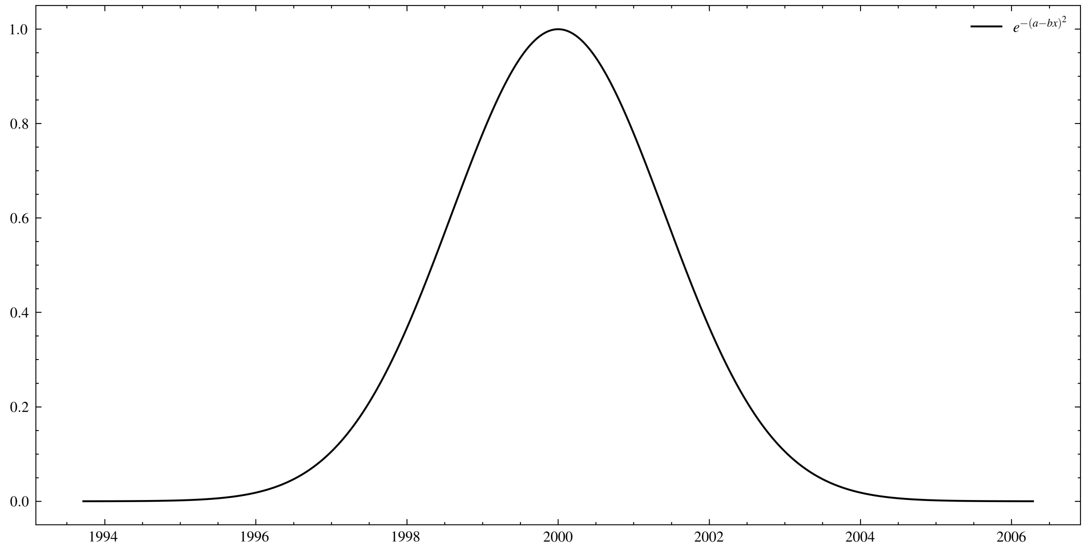
$$\int_{\frac{a-\pi}{b}}^{\frac{a+\pi}{b}} e^{-(a-bx)^2} \cdot dx$$

for,

$$a = 1000, \ b = 0.1$$

[10]:
```
a = 1000
b = 0.5


def f(x, a=a, b=b):
    return exp(-((a - b * x)**2))


x_i = (a - pi) / b
x_f = (a + pi) / b
n = 1000
```

[11]:
```
x = generate_points(x_i, x_f, n)
plt.plot(x, f(x), label="$e^{-(a-bx)^2}$")
plt.legend()
plt.show()
```




```
[12]: integrate_trapezium(f, x_i, x_f, n)
```

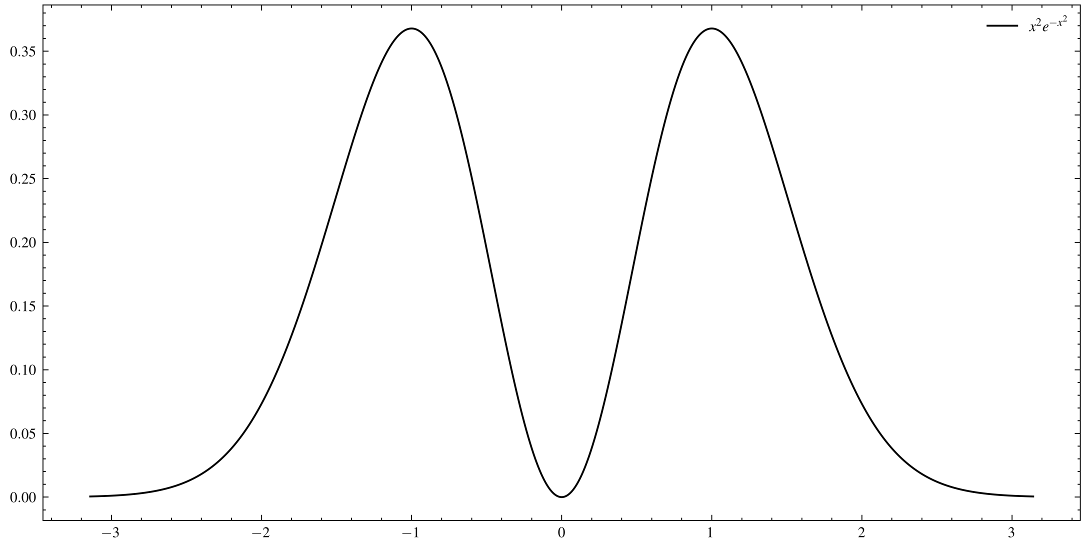
```
[12]: 3.5448768830274022
```

3.2.4

$$\int_{-\pi}^{\pi} x^2 e^{-x^2} \cdot dx$$

```
[13]: def f(x):
          return x**2 * exp(-x**2)


      x_i = - pi
      x_f = pi
      n = 1000
```

```
[14]: x = generate_points(x_i, x_f, n)
      plt.plot(x, f(x), label="$x^2e^{-x^2}$")
      plt.legend()
      plt.show()
```



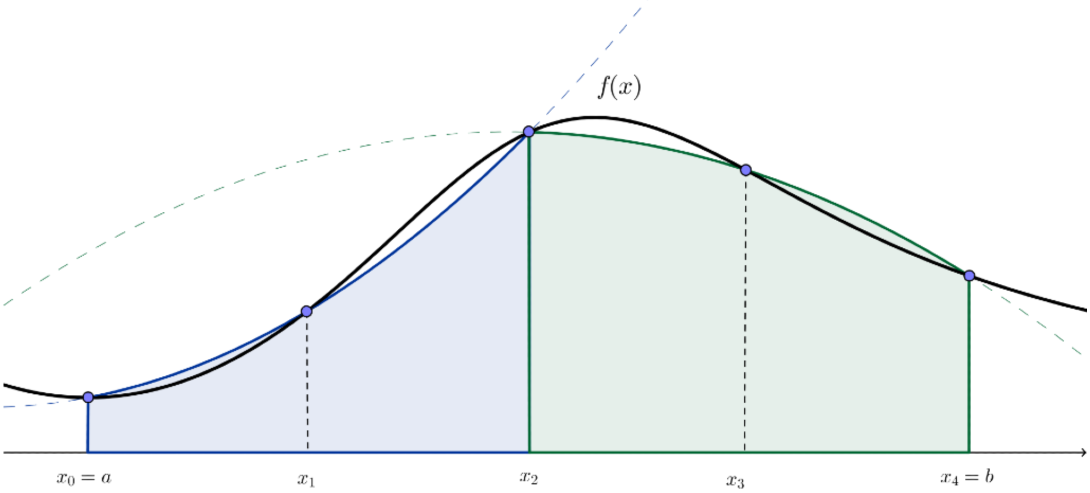
[15]:
```
integrate_trapezium(f, x_i, x_f, n)
```

[15]: 0.8860597576848352

4 Simpson's method for integration

Simpson's method for integration approximates a given curve using a number of quadratic interpolations.

The Simpson's 1/3 rule takes 3 consecutive points on the curve of the function and represents each of those points with a quadratic function to evaluate the area under the curve. This process is repeated for each points, taking 3 consecutive points at a time.



The result of the integration after evaluations is given by,

$$\int_a^b f(x)dx \approx \frac{h}{3}\left[f(x_0) + 4f(x_1) + 2f(x_2) + \cdots + 2f(x_{n-2}) + 4f(x_{n-1}) + f(x_n)\right]$$

4.1 Code

[1]:
```
from numpy import cos, empty, pi, sin, exp
from matplotlib import pyplot as plt

# Default configuaration for matplotlib
plt.style.use(['science', 'ieee'])
plt.rcParams["figure.figsize"] = (10, 5)
```

[2]:
```
# Function to generate `n` evenly spaced points between limits `a` and `b`
def generate_points(a, b, n, retstep=False):

    # Calculating the spacing (difference) between each points
    h = (b - a) / (n - 1)

    # Creating an empty array to store the points
    points = empty(n)

    # Generating the points
    for i in range(n):
        points[i] = a
        a += h

    # Returning the difference between each points if asked for
    if retstep:
        return points, h
```

```
    return points
```

[3]:
```
# Importing code for Simpson's method for integration
def integrate_simpson(f, a, b, n):

    # Checking if interation using Simpson's method is possible
    if n % 2 == 0 or n < 2:
        raise ValueError(
            "Intergration using Simpson's method can only be evaluated for odd␣
↪number of points greater than 2!")

    # Generating points
    x, h = generate_points(a, b, n, retstep=True)

    # Evaluating the function `f(x)` for each points
    y = f(x)

    # Declaring a variable to store the integral
    integral = 0

    # Evaluating the integral using Simpson's method
    for i in range(n):
        if i == 0 or i == (n - 1):
            integral += y[i]
        elif i % 2 == 0:
            integral += y[i] * 2
        elif i % 2 == 1:
            integral += y[i] * 4

    integral *= (h / 3)

    return integral
```

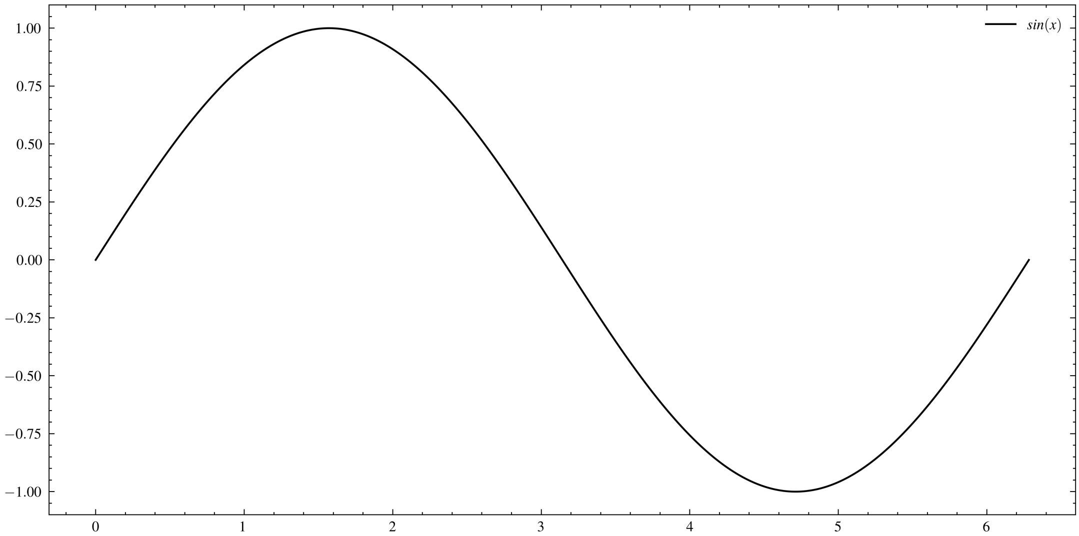
4.2 Examples

4.2.1

$$\int_0^{2\pi} sin(x) \cdot dx$$

[4]:
```
def f(x):
    return sin(x)


x_i = 0
x_f = 2 * pi
n = 1001
```

[5]:
```
x = generate_points(x_i, x_f, n)
plt.plot(x, f(x), label="$sin(x)$")
plt.legend()
plt.show()
```



[6]:
```
integrate_simpson(f, x_i, x_f, n)
```

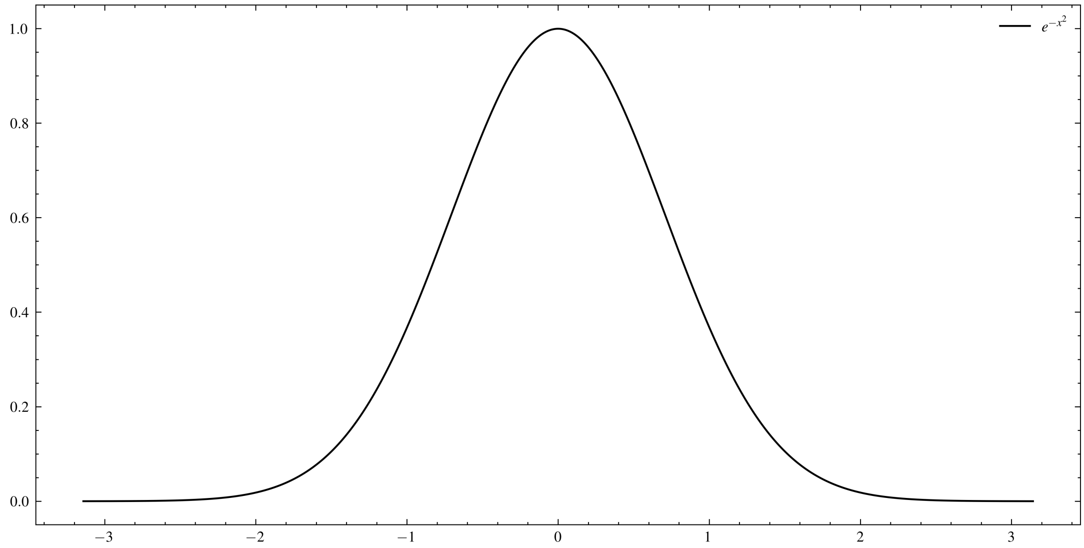
[6]: -1.1860553938477683e-13

4.2.2

$$\int_{-\pi}^{\pi} e^{-x^2} \cdot dx$$

[7]:
```
def f(x):
    return exp(-(x**2))


x_i = - pi
x_f = pi
n = 1001
```

[8]:
```
x = generate_points(x_i, x_f, n)
plt.plot(x, f(x), label="$e^{-x^2}$")
plt.legend()
plt.show()
```



[9]:
```
integrate_simpson(f, x_i, x_f, n)
```

[9]: 1.7724381183455118

4.2.3

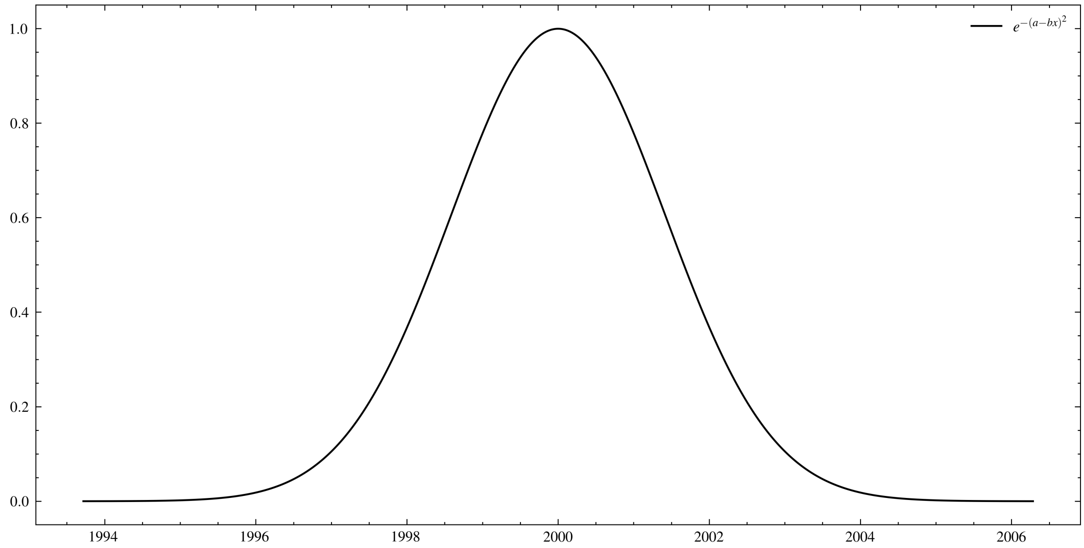
$$\int_{\frac{a-\pi}{b}}^{\frac{a+\pi}{b}} e^{-(a-bx)^2} \cdot dx$$

for,

$$a = 1000, \ b = 0.1$$

[10]:
```
a = 1000
b = 0.5


def f(x, a=a, b=b):
    return exp(-((a - b * x)**2))


x_i = (a - pi) / b
x_f = (a + pi) / b
n = 1001
```

[11]:
```
x = generate_points(x_i, x_f, n)
plt.plot(x, f(x), label="$e^{-(a-bx)^2}$")
plt.legend()
plt.show()
```




```
[12]: integrate_simpson(f, x_i, x_f, n)
```

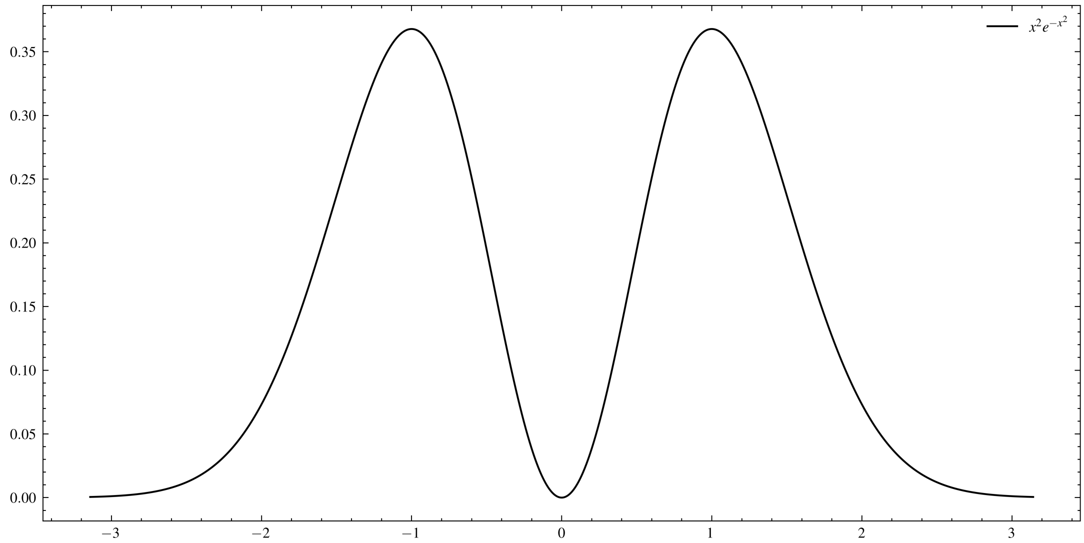
[12]: 3.544876236673206

4.2.4

$$\int_{-\pi}^{\pi} x^2 e^{-x^2} \cdot dx$$

```
[13]: def f(x):
          return x**2 * exp(-x**2)


      x_i = - pi
      x_f = pi
      n = 1001
```

```
[14]: x = generate_points(x_i, x_f, n)
      plt.plot(x, f(x), label="$x^2e^{-x^2}$")
      plt.legend()
      plt.show()
```



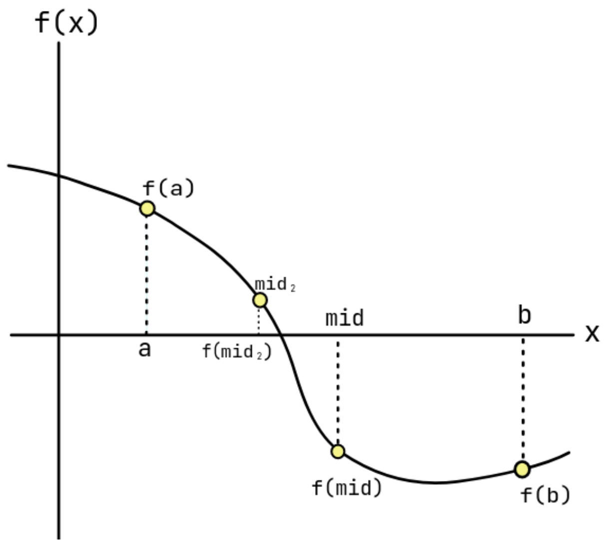
[15]:
```
integrate_simpson(f, x_i, x_f, n)
```

[15]: 0.8860565659897861

5 Bisection method for finding root

Bisection method is a root-finding algorithm that could be used to find any one root of a continuous function $f(x)$, which could be evaluated at two points a and b such that,

$$f(a) \cdot f(b) < 0$$



For the Bisection method to work we would have to provide the function $f(x)$ along with two points a and b for which the function will yeild two values, $f(a)$ and $f(b)$ of opposite signs. Then the Bisection method works by bisecting the inteval a and b, and using the midpoint, say $mid = \frac{a+b}{2}$, as a guess for the root.

If the guessed root isn't close enough to zero, then there are two possibilities,

$$f(a) \cdot f(mid) < 0$$

$$f(b) \cdot f(mid) < 0$$

For the first case b shall be shifted to the mid, and for the later a shall be shifted at the mid.

$$b = mid, \qquad if\ f(a) \cdot f(mid) < 0 \tag{1}$$
$$a = mid, \qquad if\ f(b) \cdot f(mid) < 0 \tag{2}$$

Now the next guess would be the midpoint between the updated a and b. This method is to be repeated until a suitable enough guess for the root close to zero has been found.

5.1 Code

[1]:
```
from numpy import e
```

[2]:
```
# Code for finding root using Bisection method
def find_root_bisection(f, a, b, tolerance=0.001):

    # Checking if it is possible to find the root using Bisection method
    if f(a) * f(b) >= 0:
        raise ValueError("f(a)*f(b) must be less than zero.")

    # Evaluating the root
```

```
    while abs(a - b) > tolerance:

        # Evaluating the midpoint
        mid = (a + b) / 2

        # Updating `a` and `b` as required
        if f(a) * f(mid) < 0:
            b = mid
        elif f(b) * f(mid) < 0:
            a = mid

    return mid
```

5.2 Examples

5.2.1

$$x^2 = 2$$

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

[3]:
```
def f(x):
    return x**2 - 2


a = 1
b = 5

find_root_bisection(f, a, b)
```

[3]: 1.4150390625

5.2.2

$$x^3 - x^2 + x = e$$

$$f(x) = x^3 - x^2 + x - e$$

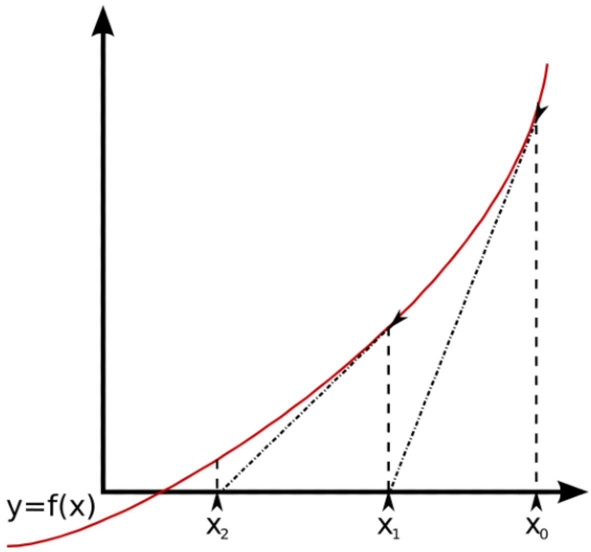
[4]:
```
def f(x):
    return x**3 - x**2 + x - e


a = 0
b = 3

find_root_bisection(f, a, b)
```

[4]: 1.519775390625

6 Newton-Raphson method for finding root

Newton-Raphson method is a root-finding algorithm to find the root (i.e., the zeroes) of a real-valued function using numerical analysis. The Newton-Raphson method produces succesive better approximations of the root with each iteration of any real function.

For finding the root of a single-variable real-valued function $f(x)$ the algorithm requires the function itself, the derivative of the function $f'(x)$, and an initial guess x_0 for the root of f.



Using $f'(x)$ we first evaluate the tangent to the curve $f(x)$ at $x = x_0$. Then, our next guess x_1, becomes the intersection of this tangent at the $x-axis$. From the diagram above, the derivetive of $f(x)$ at $x = x_0$ will be given by,

$$f'(x_0) = \frac{f(x_0) - 0}{x_0 - x_1}$$

Thus, the initial guess x_1 would be given by,

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

The next guess x_2 would be,

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)}$$

This process is to be repeated again and again until a guess close enough to the actual root is attained.

Therefore, the generic form of the nth guess for the root would be,

$$x_n = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})}$$

6.1 Code

[1]:
```
from numpy import cos, e, linspace, pi, sin
```

[2]:
```
# Code for finding root using Newton-Raphson method
def find_root_newton(f, f_prime, x_0, tolerance=0.001):

    # Declaring a variable to store the `f(x) / f_prime(x)`
    h = (f(x_0) / f_prime(x_0))

```

```python
    # Evaluating the root
    while abs(h) > tolerance:

        # Evaluating `f(x) / f_prime(x)`
        h = (f(x_0) / f_prime(x_0))

        # Subtracting `f(x) / f_prime(x)` from the previous guess
        x_0 -= h

    return x_0
```

6.2 Examples

6.2.1

$$x^2 = 2$$

$$f(x) = x^2 - 2$$
$$f'(x) = 2x$$

[3]:
```python
def f(x):
    return x**2 - 2


def f_prime(x):
    return 2 * x


x_0 = pi

find_root_newton(f, f_prime, x_0)
```

[3]: 1.414213562373189

6.2.2

$$x^3 - x^2 + x = e$$

$$f(x) = x^3 - x^2 + x - e$$
$$f'(x) = 3x^2 - 2x + 1$$

[4]:
```python
def f(x):
    return x**3 - x**2 + x - e


def f_prime(x):
    return 3 * x**2 - 2 * x + 1


x_0 = 0

find_root_newton(f, f_prime, x_0)
```

[4]: 1.5193605629027016

7 Fourier series

Any periodic function can be expressed as a summation of an infinite series of trigionometric (sine and cosine) terms, such a representation is called a Fourier series.

The most common form a Fourier series is,

$$f(x) = \frac{a_0}{2} + \sum_{n=1}^{\infty} [a_n \cdot cos(nx) + b_n \cdot sin(nx)]$$

which interpolates a periodic function of period 2π.

For a generic periodic function $f(x)$, having period $2l$, the Fourier series is represented as,

$$f(x) = \frac{a_0}{2} + \sum_{n=1}^{\infty} \left[a_n \cdot cos\left(\frac{n\pi}{l}x\right) + b_n \cdot sin\left(\frac{n\pi}{l}x\right)\right]$$

The coefficients a_n and b_n to the sine and cosine terms, can be eveluated as follows,

$$a_n = \frac{1}{l}\int_{-l}^{l} f(x) \cdot cos\left(\frac{n\pi}{l}x\right) dx$$

$$b_n = \frac{1}{l}\int_{-l}^{l} f(x) \cdot sin\left(\frac{n\pi}{l}x\right) dx$$

7.1 Code

[1]:
```
from numpy import cos, empty, pi, sin, vectorize
from matplotlib import pyplot as plt

# Default configuaration for matplotlib
plt.style.use(['science', 'ieee'])
plt.rcParams["figure.figsize"] = (10, 5)
```

[2]:
```
# Function to generate `n` evenly spaced points between limits `a` and `b`
def generate_points(a, b, n, retstep=False):

    # Calculating the spacing (difference) between each points
    h = (b - a) / (n - 1)

    # Creating an empty array to store the points
    points = empty(n)

    # Generating the points
    for i in range(n):
        points[i] = a
        a += h

    # Returning the difference between each points if asked for
    if retstep:
        return points, h

    return points
```

[3]:
```
# Importing code for Simpson's method for integration
def integrate_simpson(f, x_i, x_f, n):

    # Checking if interation using Simpson's method is possible
    if n % 2 == 0 or n < 2:
```

```
        raise ValueError(
            "Intergration using Simpson's method can only be evaluated for odd␣
↪number of points greater than 2!")

    # Generating points
    x, h = generate_points(x_i, x_f, n, retstep=True)

    # Evaluating the function `f(x)` for each points
    y = f(x)

    # Declaring a variable to store the integral
    integral = 0

    # Evaluating the integral using Simpson's method
    for i in range(n):
        if i == 0 or i == (n - 1):
            integral += y[i]
        elif i % 2 == 0:
            integral += y[i] * 2
        elif i % 2 == 1:
            integral += y[i] * 4

    integral *= (h / 3)

    return integral
```

[4]:
```
# Code for evaluating coefficients to the sine and cosine terms of Fourier series
def evaluate_fourier_coeff(f, a, b, n, integration_points=10**3+1):

    # Calculating the half-period of the fuction and storing it
    half_period = 0.5 * (b - a)

    # Creating an empty array to store the Fourier coefficients
    fourier_coeff = empty((n, 2))

    # Evaluating the Fourier coefficients
    for i in range(n):
        fourier_coeff[i][0] = (1 / half_period) * integrate_simpson(
            (lambda x: f(x) * cos(i * (pi * x / half_period))), a, b,␣
↪integration_points)
        fourier_coeff[i][1] = (1 / half_period) * integrate_simpson(
            (lambda x: f(x) * sin(i * (pi * x / half_period))), a, b,␣
↪integration_points)

    return fourier_coeff
```

[5]:
```
# Function to evaluate Fourier approximation using the Fourier coefficients
def evaluate_fourier_approx(x, fourier_coeff, half_period):

    # Declaring a variable to store the Fourier approximation and initialising it␣
↪with 0.5 times the first Fourier cosine coeficient `a_0`
    fourier_approx = fourier_coeff[0][0] * 0.5

    # Evaluating the Fourier approximation using the Fourier coefficients
    for i in range(1, fourier_coeff.shape[0]):
        fourier_approx += (fourier_coeff[i][0] * cos(i * (pi * x / half_period)) +␣
↪(
```

```
            fourier_coeff[i][1] * sin(i * (pi * x / half_period)))

    return fourier_approx
```

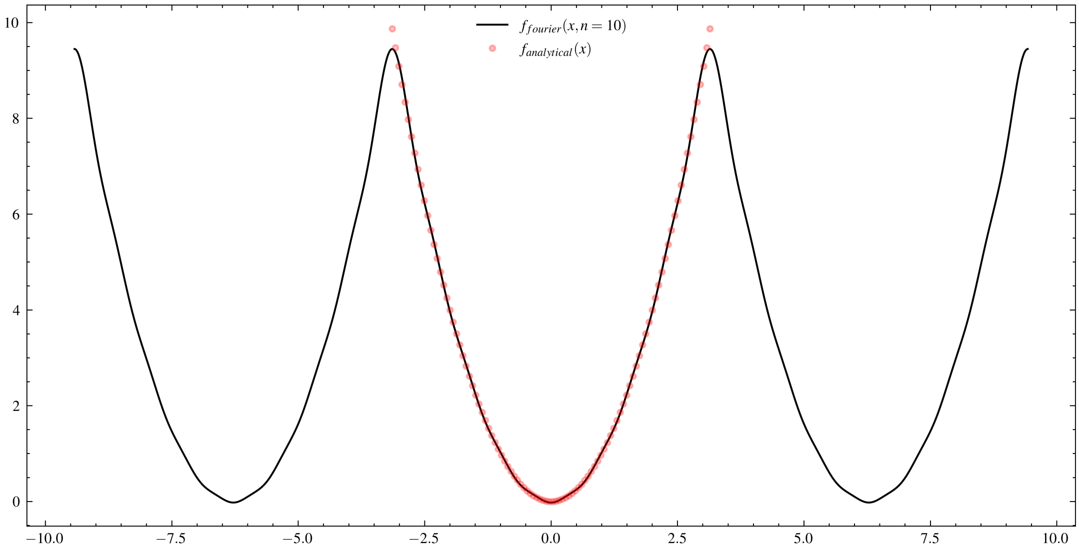
[6]:
```
# Fuction to evaluate and plot Fourier approximation
def fourier_plot(f, a, b, n_coeffcients, plotting_a, plotting_b,␣
 ↪plotting_points=10**3+1):

    # Generating points for plotting Fourier approximation
    points = generate_points(plotting_a, plotting_b, plotting_points)

    # Evaluating the maximum number of Fourier coefficients required for plot
    fourier_coeff = evaluate_fourier_coeff(f, a, b, max(n_coeffcients))

    # Plotting the Fourier approximation for different number of Fourier␣
 ↪coefficients
    for n in n_coeffcients:

        # Evaluating the Fourier approximation using different number of Fourier␣
 ↪coefficients
        fourier_approx = evaluate_fourier_approx(
            points, fourier_coeff[:n], 0.5 * (b - a))

        # Ploting the Fourier approximation
        plt.plot(points, fourier_approx, label=f"$f_{{fourier}}(x, n = {n})$")

    # Ploting the original function in its period
    points_period = generate_points(a, b, plotting_points // 10)
    plt.plot(points_period, f(points_period), ".",
             alpha=0.25, label="$f_{analytical}(x)$")

    plt.legend()
    plt.show()
```

7.2 Examples

7.2.1

$$f(x) = x^2,\ x \in [-\pi, \pi]$$

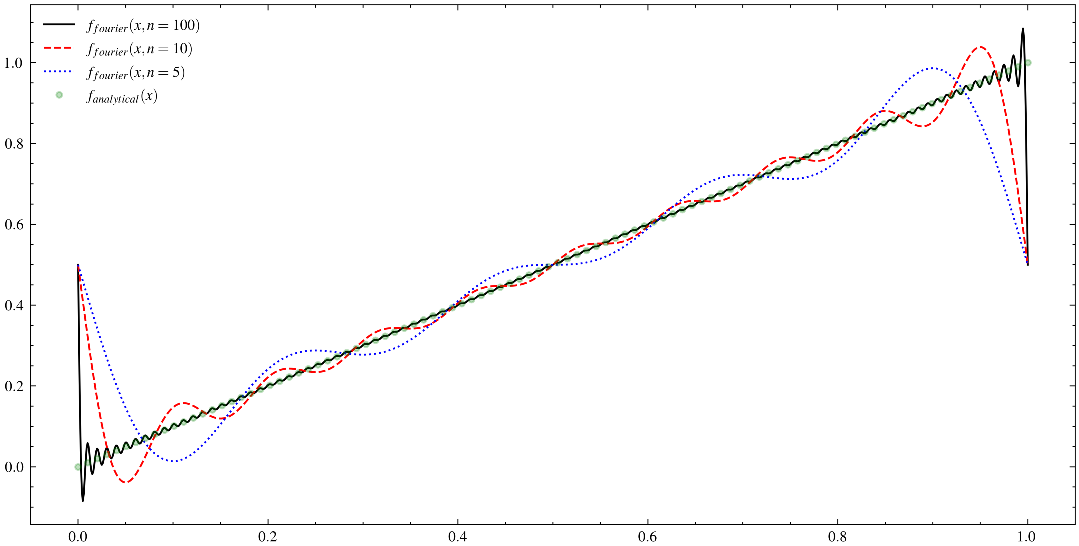
[7]:
```
def f(x):
    return x**2


a = - pi
b = pi
n_coeffcients = [10]

fourier_plot(f, a, b, n_coeffcients, 3*a, 3*b)
```



7.2.2

$$f(x) = x, \;\; x \in [0, 1]$$

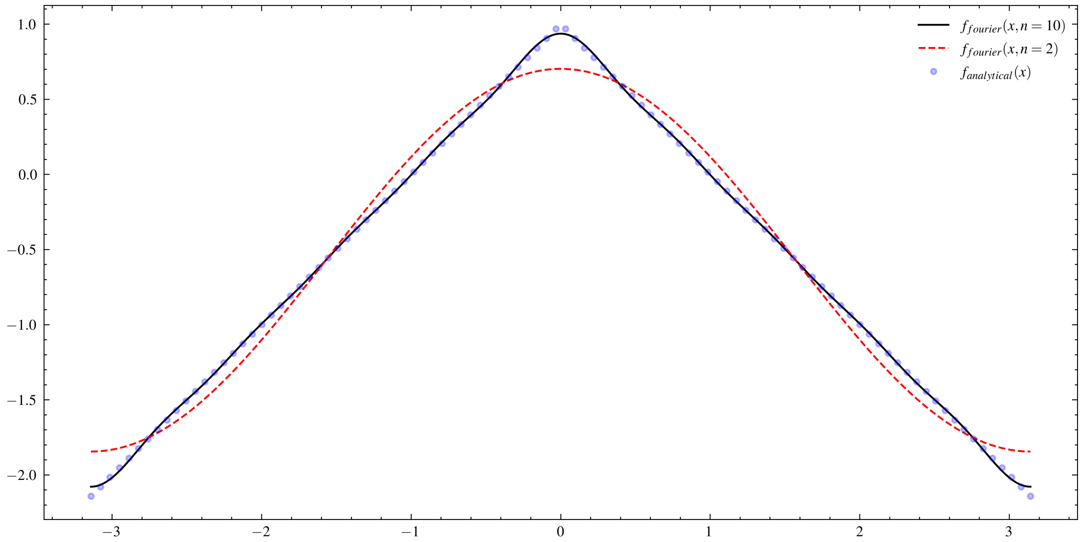
[8]:
```python
def f(x):
    return x


a = 0
b = 1
n_coeffcients = [100, 10, 5]

fourier_plot(f, a, b, n_coeffcients, a, b)
```



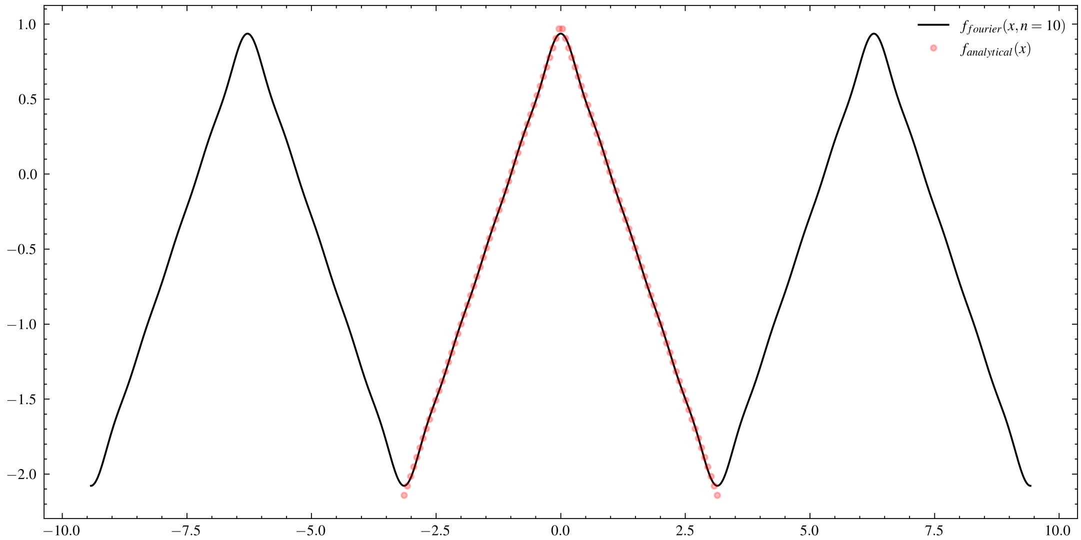
7.2.3

$$f(x) = 1 - |x|, \;\; x \in [-\pi, \pi]$$

[9]:
```
def f(x):
    return 1 - abs(x)


a = - pi
b = pi
n_coeffcients = [10, 2]

fourier_plot(f, a, b, n_coeffcients, a, b)
```



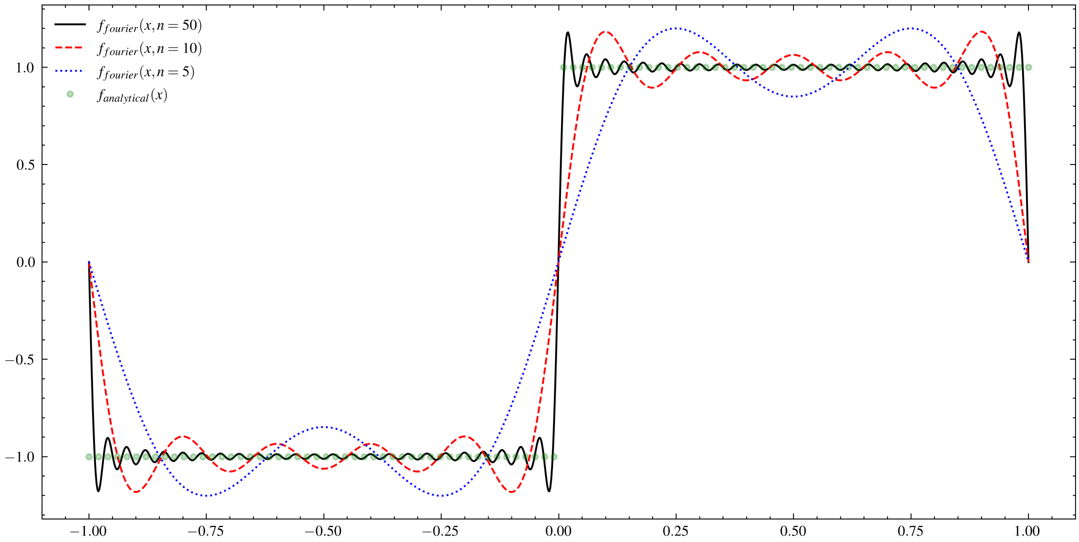
[10]:
```
n_coeffcients = [10]

fourier_plot(f, a, b, n_coeffcients, 3*a, 3*b)
```



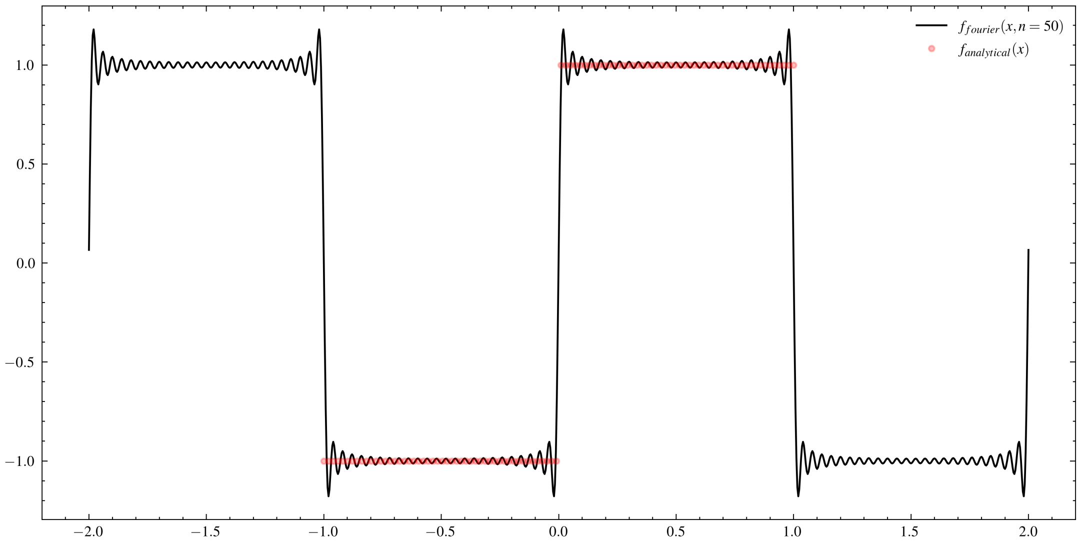
7.2.4

$$f(x) = \begin{cases} 1, & x \in [-1, 0) \\ -1, & x \in [0, 1] \end{cases}$$

[11]:
```
def f(x):
    if x > 0:
        return 1
    else:
        return -1


f = vectorize(f)
a = - 1
b = 1
n_coeffcients = [50, 10, 5]

fourier_plot(f, a, b, n_coeffcients, a, b)
```



[12]:
```
n_coeffcients = [50]

fourier_plot(f, a, b, n_coeffcients, 2*a, 2*b)
```



8 Fourier transformation

A Fourier transform (FT) decomposes a periodic function into its frequency components. The Fourier transform $g(\alpha)$ of a given function $f(x)$ is given by,

$$g(\alpha) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(x) e^{-i\alpha x} dx$$

The invert Fourier transforms of the fuction $g(\alpha)$ in turn gives the original function back,

$$f(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} g(\alpha) e^{i\alpha x} d\alpha$$

These two functions $g(\alpha)$ and $f(x)$ are called a pair of Fourier transforms.

By using the Euler's formula, we can write the first equation as,

$$g(\alpha) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(x) \left[cos(\alpha x) - i\ sin(\alpha x)\right] dx$$

$$g(\alpha) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(x) cos(\alpha x) dx - \frac{i}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(x) sin(\alpha x) dx$$

Let,

$$u(\alpha) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(x) cos(\alpha x) dx$$

$$v(\alpha) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(x) sin(\alpha x) dx$$

Therefore,

$$g(\alpha) = u(\alpha) - i\ v(\alpha)$$

$$FT(\alpha) = FT_{cos}(\alpha) - i\ FT_{sin}(\alpha)$$

Now, we can evalute the functions $u(\alpha)$ and $v(\alpha)$ seperately to get the fourier transform for $f(x)$.

8.1 Code

[1]:
```python
from numpy import cos, empty, exp, pi, sin, sqrt, vectorize
from matplotlib import pyplot as plt

# Default configuaration for matplotlib
plt.style.use(['science', 'ieee'])
plt.rcParams["figure.figsize"] = (10, 5)
```

[2]:
```python
# Function to generate `n` evenly spaced points between limits `a` and `b`
def generate_points(a, b, n, retstep=False):

    # Calculating the spacing (difference) between each points
    h = (b - a) / (n - 1)

    # Creating an empty array to store the points
    points = empty(n)

    # Generating the points
    for i in range(n):
        points[i] = a
        a += h
```

```
    # Returning the difference between each points if asked for
    if retstep:
        return points, h

    return points
```

[3]:
```
# Importing code for Simpson's method for integration
def integrate_simpson(f, x_i, x_f, n):

    # Checking if interation using Simpson's method is possible
    if n % 2 == 0 or n < 2:
        raise ValueError(
            "Intergration using Simpson's method can only be evaluated for odd␣
↪number of points greater than 2!")

    # Generating points
    x, h = generate_points(x_i, x_f, n, retstep=True)

    # Evaluating the function `f(x)` for each points
    y = f(x)

    # Declaring a variable to store the integral
    integral = 0

    # Evaluating the integral using Simpson's method
    for i in range(n):
        if i == 0 or i == (n - 1):
            integral += y[i]
        elif i % 2 == 0:
            integral += y[i] * 2
        elif i % 2 == 1:
            integral += y[i] * 4

    integral *= (h / 3)

    return integral
```

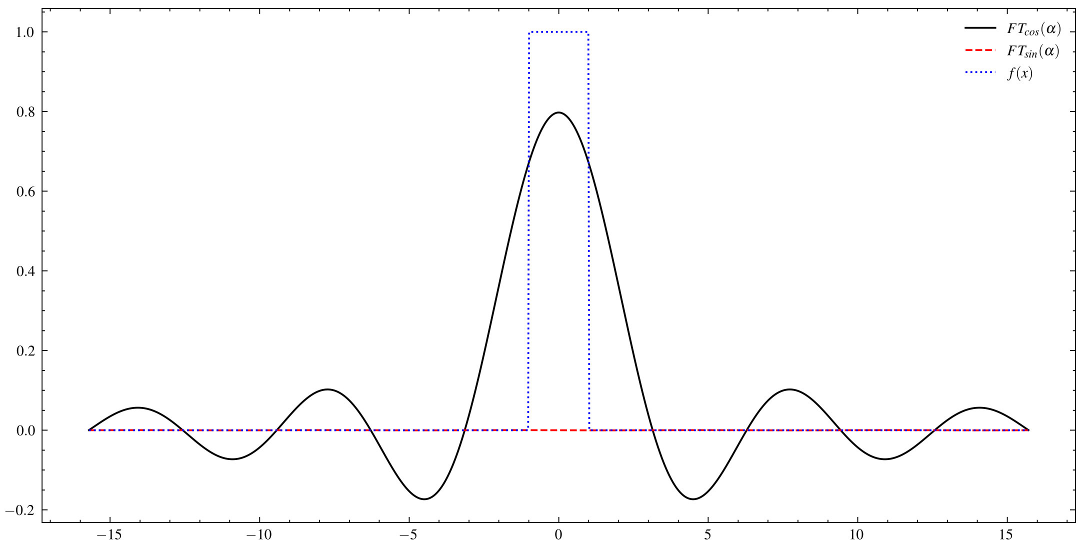
[4]:
```
# Code for evaluating Fourier transform
def fourier_transform(f, a, b, integration_points=10**3+1):

    # Function for evaluating the cosine term of the Fourier transform
    def u(alpha):
        return (1 / sqrt(2 * pi)) * integrate_simpson(lambda x: f(x) * cos(alpha *␣
↪x), a, b, integration_points)

    # Function for evaluating the sin term of the Fourier transform
    def v(alpha):
        return (1 / sqrt(2 * pi)) * integrate_simpson(lambda x: f(x) * sin(alpha *␣
↪x), a, b, integration_points)

    return vectorize(u), vectorize(v)
```

8.2 Examples

8.2.1

$$f(x) = \begin{cases} 1, & x \in [-1, 1] \\ 0, & x \in [else] \end{cases}$$

[5]:
```python
def f(x):
    if abs(x) <= 1:
        return 1.0
    return 0.0


f = vectorize(f)
a = - 1
b = 1

ft_u, ft_v = fourier_transform(f, a, b)

x_i = - 5 * pi
x_f = 5 * pi
n = 1000

x = generate_points(x_i, x_f, n)

plt.plot(x, ft_u(x), label="$FT_{cos}(\\alpha)$")
plt.plot(x, ft_v(x), label="$FT_{sin}(\\alpha)$")
plt.plot(x, f(x), label="$f(x)$")
plt.legend()
plt.show()
```

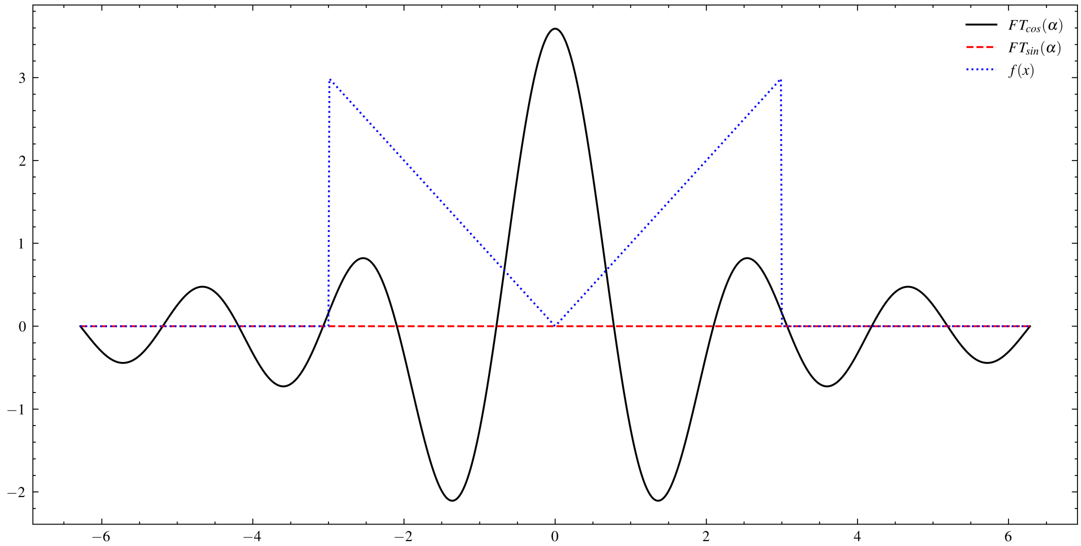


8.2.2

$$f(x) = \begin{cases} 1 - |x|, & x \in [-3, 3] \\ 0, & x \in [else] \end{cases}$$

[6]:
```python
def f(x):
    if abs(x) <= 3:
        return abs(x)
    return 0.0


f = vectorize(f)
a = - 3
b = 3
```

```
ft_u, ft_v = fourier_transform(f, a, b)

x_i = - 2 * pi
x_f = 2 * pi
n = 1000

x = generate_points(x_i, x_f, n)

plt.plot(x, ft_u(x), label="$FT_{cos}(\\alpha)$")
plt.plot(x, ft_v(x), label="$FT_{sin}(\\alpha)$")
plt.plot(x, f(x), label="$f(x)$")
plt.legend()
plt.show()
```

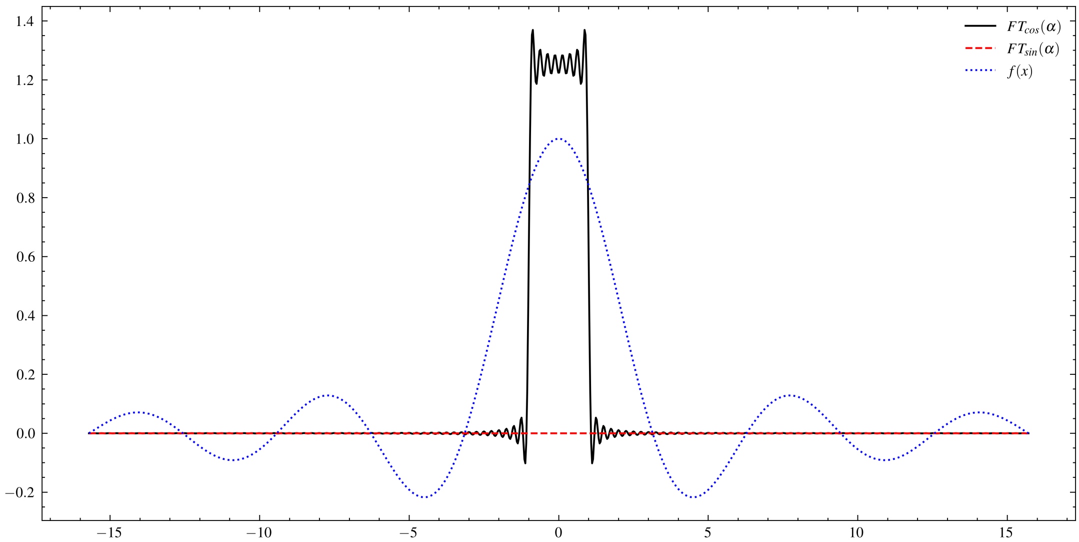


8.2.3

$$f(x) = \frac{sin(x)}{x}$$

[7]:
```
def f(x):
    return sin(x) / x


a = - 25
b = 25

ft_u, ft_v = fourier_transform(f, a, b)

x_i = - 5 * pi
x_f = 5 * pi
n = 1000

x = generate_points(x_i, x_f, n)

plt.plot(x, ft_u(x), label="$FT_{cos}(\\alpha)$")
plt.plot(x, ft_v(x), label="$FT_{sin}(\\alpha)$")
plt.plot(x, f(x), label="$f(x)$")
```

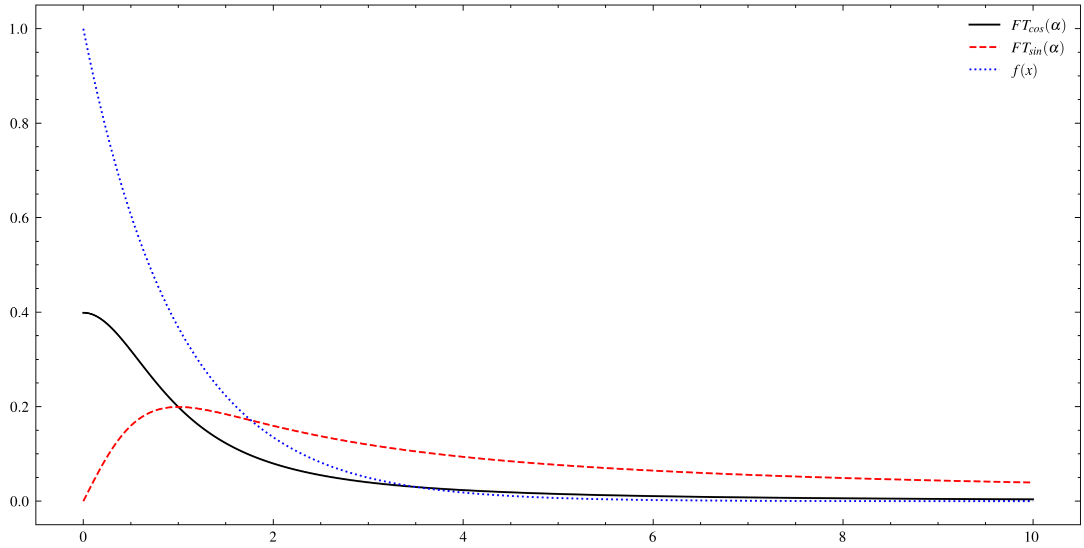
```
plt.legend()
plt.show()
```



8.2.4

$$f(x) = e^{-x}, \;\; x \in [0, \infty)$$

[8]:
```
def f(x):
    if x >= 0:
        return exp(-x)
    return 0.0


f = vectorize(f)
a = 0
b = 10

ft_u, ft_v = fourier_transform(f, a, b)

x_i = 0
x_f = 10
n = 1000

x = generate_points(x_i, x_f, n)

plt.plot(x, ft_u(x), label="$FT_{cos}(\\alpha)$")
plt.plot(x, ft_v(x), label="$FT_{sin}(\\alpha)$")
plt.plot(x, f(x), label="$f(x)$")
plt.legend()
plt.show()
```

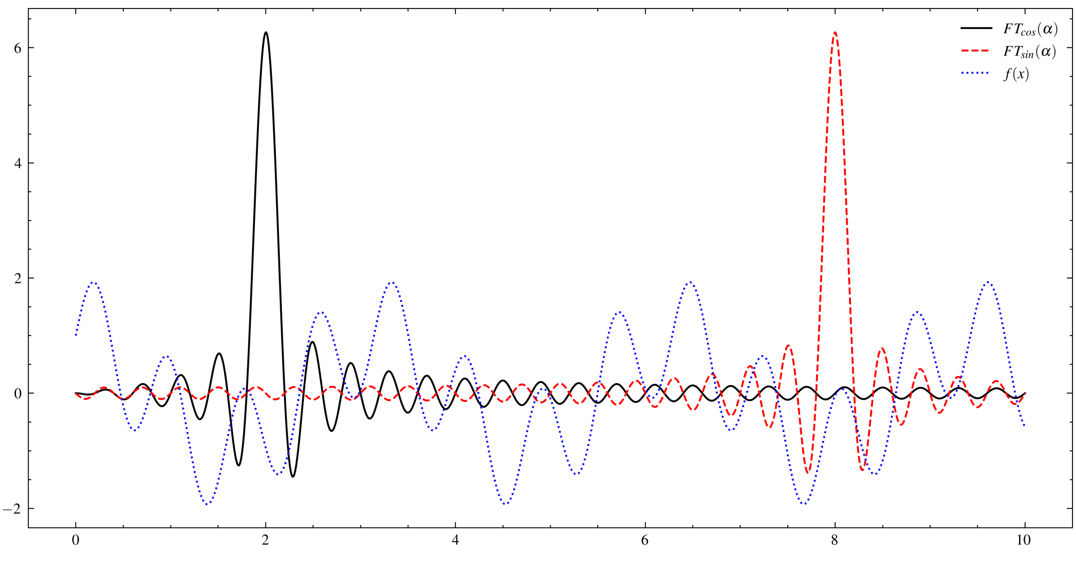


8.2.5

$$f(x) = cos(2x) + sin(8x)$$

Note: Deliberately mentioning the frequency of the periodic function to verify the results of the Fourier transforms

[9]:
```
def f(x):
    return cos(2*x) + sin(8*x)


a = - 5 * pi
b = 5 * pi

ft_u, ft_v = fourier_transform(f, a, b)

x_i = 0
x_f = 10
n = 1000

x = generate_points(x_i, x_f, n)

plt.plot(x, ft_u(x), label="$FT_{cos}(\\alpha)$")
plt.plot(x, ft_v(x), label="$FT_{sin}(\\alpha)$")
plt.plot(x, f(x), label="$f(x)$")
plt.legend()
plt.show()
```



$FT_{cos}(\alpha)$
$FT_{sin}(\alpha)$
$f(x)$
6
4
2
0
−2
0
2
4
6
8
10

9 Euler method for solving first-order differential equations

The Euler method is a numerical algorithm for solving ordinary differential equations given an initial value.

Given an initial value, the method works by evaluating the function $f(x)$ at a point $x = x_{n+1}$ using straight line approximation with slope $f'(x)$ at $x = x_n$, where the step, $h = x_{n+1} - x_n$.

Using the first principle of derivatives,

$$f'(x_0) = \lim_{h \to 0} \frac{f(x_0 + h) - f(x_0)}{h}$$

Therfore, if h is small enough,

$$f(x_0 + h) \approx f(x_0) + h \cdot f'(x_0)$$

Another possible explanation of this approximation, can originate from the Taylor expansion,

$$f(x_0 + h) = f(x_0) + h \cdot f'(x_0) + h^2 \cdot f''(x_0) + h^3 \cdot f'''(x_0) + ...$$

Since $h << 1$, ignoring the higher order terms the approximation becomes,

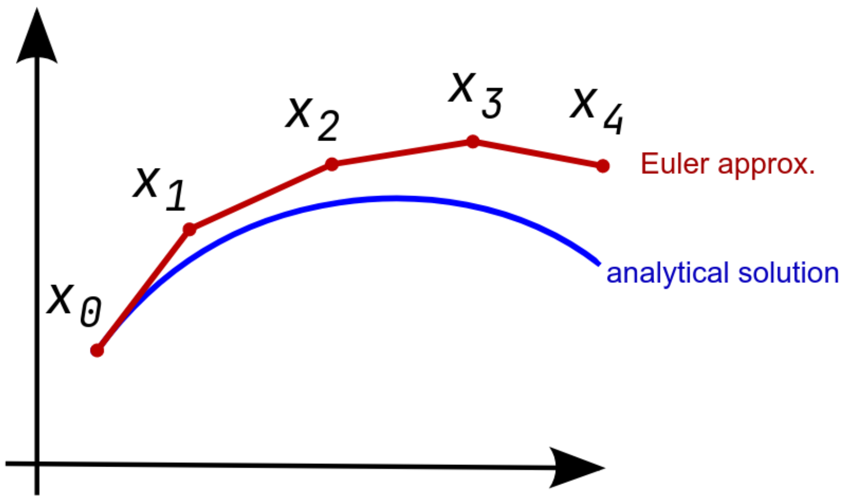
$$f(x_0 + h) \approx f(x_0) + h \cdot f'(x_0)$$

Then the function $f(x)$ at a generic point x_n can be approximated using,

$$f(x_n) \approx f(x_{n-1}) + h \cdot f'(x_{n-1})$$

The intial value of $f(x)$ at $x = x_0$ must be provided, which will be the starting point for the Euler method algorithm. This will eliminate the constant we would have got from solving the first-order differential equation.

An Illustration of the Euler method is given below,



9.1 Code

[1]:
```
from numpy import cos, empty, exp, pi, sin
from matplotlib import pyplot as plt

# Default configuaration for matplotlib
plt.style.use(['science', 'ieee'])
plt.rcParams["figure.figsize"] = (10, 5)
```

[2]:
```python
# Function to generate `n` evenly spaced points between limits `a` and `b`
def generate_points(a, b, h):

    # Calculating the number of points
    n = int(((b - a) / h) + 1)

    # Creating an empty array to store the points
    points = empty(n)

    # Generating the points
    for i in range(n):
        points[i] = a
        a += h

    return points
```

[3]:
```python
# Code for solving first-order ordinary differential equations using Euler method
def solve_euler(y_prime, x_i, x_f, y_i, h):

    # Generating equispaced points
    x = generate_points(x_i, x_f, h)

    # Creating a array to store `y`
    y = empty(x.size)

    # Initialising `y` with initial value
    y[0] = y_i

    # Evaluating the function `y` at the remaining points
    for i in range(1, x.size):
        y[i] = y[i-1] + h * y_prime(x[i], y[i-1])

    return x, y
```

9.2 Examples

9.2.1

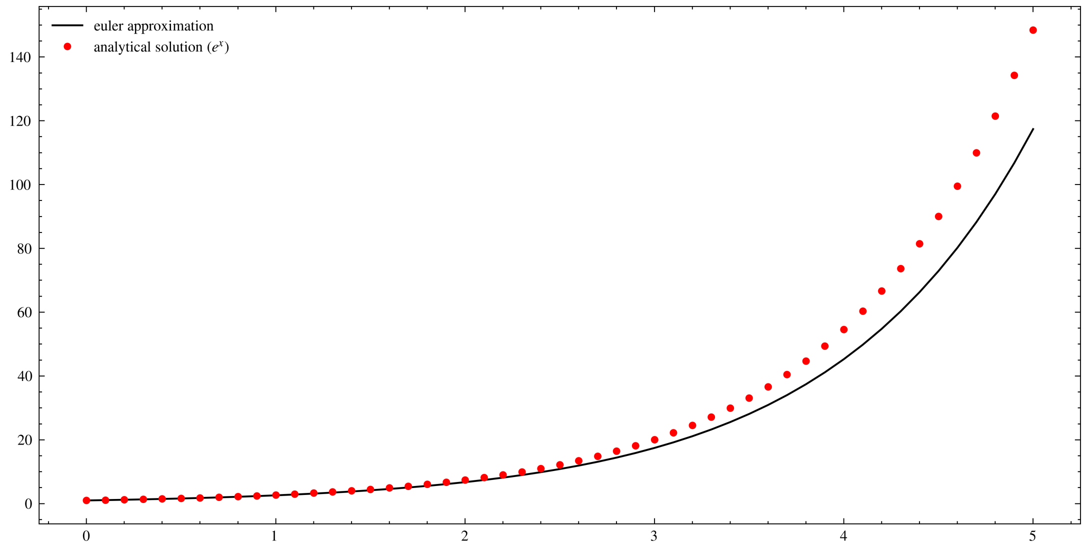
$$\frac{dy}{dx} = y$$

[4]:
```python
def y_prime(x, y):
    return y


x_i = 0
x_f = 5
y_i = 1
h = 0.1
```

[5]:
```python
# Euler approximation
points, y_euler = solve_euler(y_prime, x_i, x_f, y_i, h)

# Analytical solution
y_analytical = exp(points)
```

[6]:
```python
plt.plot(points, y_euler, label="euler approximation")
plt.plot(points, y_analytical, '.', label="analytical solution ($e^x$)")
```

```
plt.legend()
plt.show()
```



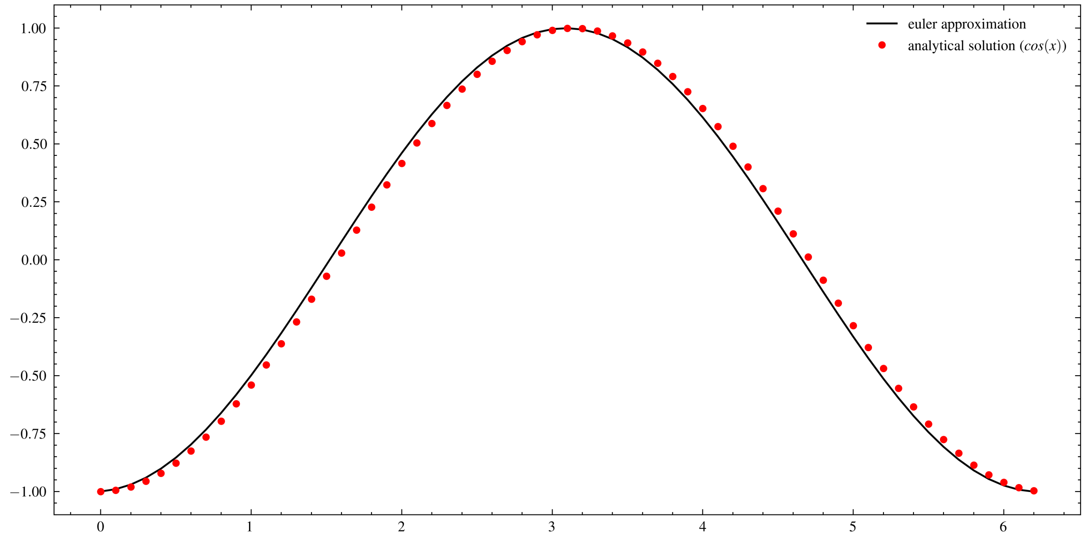
9.2.2

$$\frac{dy}{dx} = sin(x)$$

[7]:
```
def y_prime(x, y):
    return sin(x)


x_i = 0
x_f = 2 * pi
y_i = - 1
h = 0.1
```

[8]:
```
# Euler approximation
points, y_euler = solve_euler(y_prime, x_i, x_f, y_i, h)

# Analytical solution
y_analytical = -cos(points)
```

[9]:
```
plt.plot(points, y_euler, label="euler approximation")
plt.plot(points, y_analytical, '.', label="analytical solution ($cos(x)$)")
plt.legend()
plt.show()
```


euler approximation
analytical solution ($cos(x)$)
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
1
2
3
4
5
6

10 Euler method for solving second-order differential equations

Similar to the first-order Euler method, the second-order differential equations can also solved using the Euler method.

Previously we got,

$$f(x_n) \approx f(x_{n-1}) + h \cdot f'(x_{n-1})$$

Replacing $f(x_n)$ with its derivative,

$$f'(x_n) \approx f'(x_{n-1}) + h \cdot f''(x_{n-1})$$

Putting this equation into the previous one,

$$f(x_n) \approx f(x_{n-1}) + h\left[f'(x_{n-1}) + h \cdot f''(x_{n-1})\right]$$

$$f(x_n) \approx f(x_{n-1}) + h \cdot f'(x_{n-1}) + h^2 \cdot f''(x_{n-1})$$

Again we can also reach here from the Taylor expansion, but this time we have to consider upto the third term,

$$f(x_n) \approx f(x_{n-1}) + h \cdot f'(x_{n-1}) + h^2 \cdot f''(x_{n-1})$$

But while writing the code we will first eveluate the first derivaitve using the euler method and then again use the first derivetive to evalute the function itsef.

$$f(x_n) = f(x_{n-1}) + h \cdot f'(x_{n-1})$$

$$f'(x_n) = f'(x_{n-1}) + h \cdot f''(x_{n-1})$$

For this we would not only have to provide the initial value of $f(x)$, but also the intial value of $f'(x)$ both at $x = x_0$, which will be starting point for the Euler algorithm when solving for $f(x)$ and $f'(x)$ respectively. This would also eliminate the two constants we would have got from solving the second-order differential equations.

10.1 Code

[1]:
```
from numpy import cos, empty, exp, pi, sin
from scipy.constants import g
from matplotlib import pyplot as plt

# Default configuaration for matplotlib
plt.style.use(['science', 'ieee'])
plt.rcParams["figure.figsize"] = (10, 5)
```

[2]:
```
# Function to generate `n` evenly spaced points between limits `a` and `b`
def generate_points(a, b, h):

    # Calculating the number of points
    n = int(((b - a) / h) + 1)

    # Creating an empty array to store the points
    points = empty(n)

    # Generating the points
    for i in range(n):
        points[i] = a
        a += h

    return points
```

[3]:
```
# Code for solving second-order ordinary differential equations using Euler method
def solve_euler(y_prime_prime, x_i, x_f, y_i, y_prime_i, h):

    # Generating equispaced points
    x = generate_points(x_i, x_f, h)

    # Creating two arrays to store `y_prime` and `y`
    y_prime = empty(x.size)
    y = empty(x.size)

    # Initialising `y_prime` and `y` with initial values
    y_prime[0] = y_prime_i
    y[0] = y_i

    # Evaluating the function `y_prime` and `y` at the remaining points
    for i in range(1, x.size):
        y_prime[i] = y_prime[i-1] + h * \
            y_prime_prime(x[i], y[i-1], y_prime[i-1])
        y[i] = y[i-1] + h * y_prime[i-1]

    return x, y
```

10.2 Examples

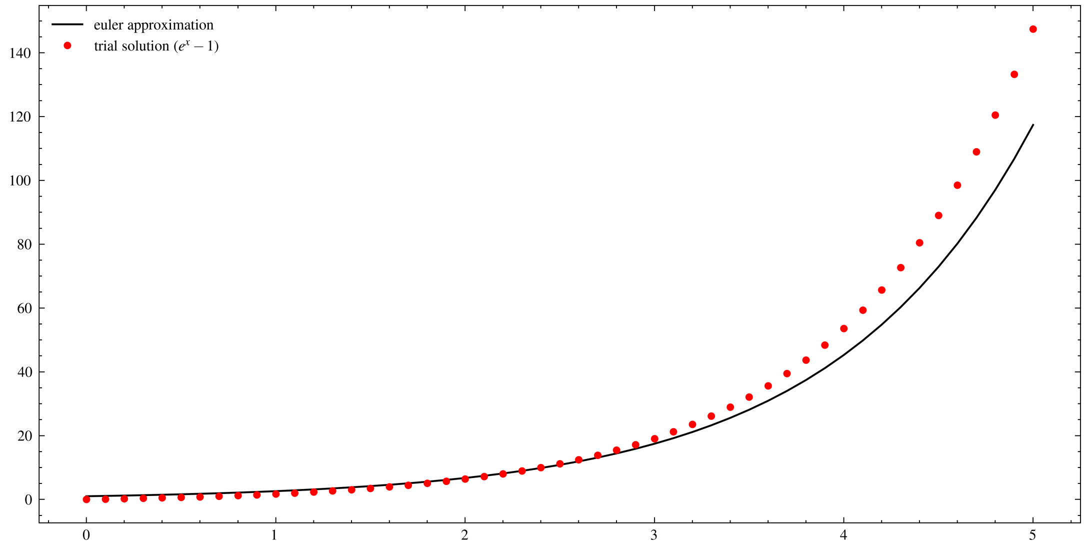
10.2.1

$$\frac{d^2y}{dx^2} = y$$

[4]:
```
def y_prime_prime(x, y, y_prime):
    return y


x_i = 0
x_f = 5
h = 0.1

y_i = 1
y_prime_i = 1
```

[5]:
```
# Euler approximation
points, y_euler = solve_euler(y_prime_prime, x_i, x_f, y_i, y_prime_i, h)

# Trial solution
y_trial = exp(points) - 1
```

[6]:
```
plt.plot(points, y_euler, label="euler approximation")
plt.plot(points, y_trial, '.', label="trial solution ($e^x - 1$)")
plt.legend()
plt.show()
```



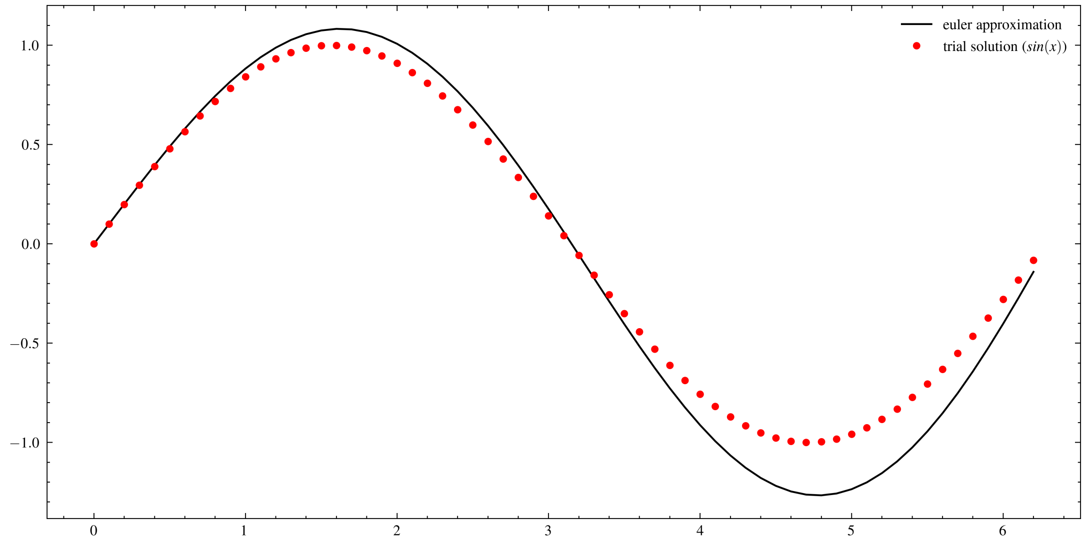
10.2.2

$$\frac{d^2y}{dx^2} = -y$$

[7]:
```
def y_prime_prime(x, y, y_prime):
    return -y


x_i = 0
x_f = 2 * pi
h = 0.1

y_i = 0
y_prime_i = 1
```

[8]:
```
# Euler approximation
points, y_euler = solve_euler(y_prime_prime, x_i, x_f, y_i, y_prime_i, h)

# Trial solution
y_trial = sin(points)
```

[9]:
```
plt.plot(points, y_euler, label="euler approximation")
plt.plot(points, y_trial, '.', label="trial solution ($sin(x)$)")
plt.legend()
plt.show()
```



10.3 Real world physics problems

10.3.1 Harmonic oscillator

The equation of motion for an ideal harmonic oscillator is,

$$m\frac{d^2x}{dt^2} = -kx$$

$$m\ddot{x} = -kx$$

In presence of the force of friction proprtional to the velocity ($\dot{x}$) of the harmonic oscillator the equation becomes,

$$m\ddot{x} = -kx - b\dot{x}$$

Which can further be expressed in the standard form,

$$\ddot{x} + 2\gamma\dot{x} + \omega^2 x = 0$$

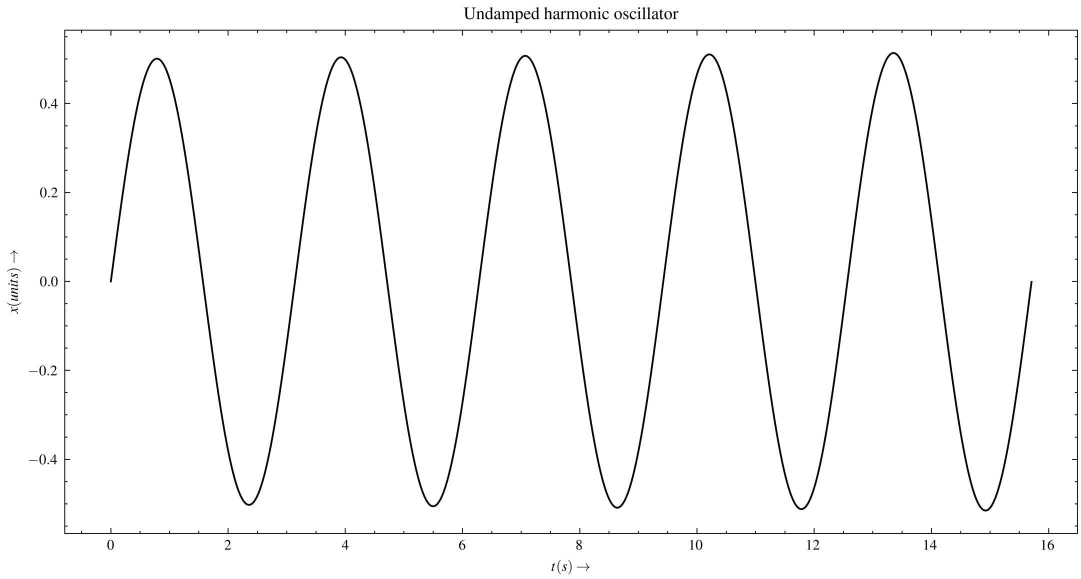
where, $\gamma = b/2m$ and $\omega^2 = k/m$.

Undamped harmonic oscillator

$$\gamma = 0$$

[10]:
```
gamma = 0
omega = 2


def x_prime_prime(t, x, x_prime):
    return - 2 * gamma * x_prime - omega**2 * x


t_i = 0
t_f = 5 * pi
x_i = 0
x_prime_i = 1
h = 0.001
```

[11]:
```python
points, x_undamp = solve_euler(x_prime_prime, t_i, t_f, x_i, x_prime_i, h)

plt.plot(points, x_undamp)
plt.title("Undamped harmonic oscillator")
plt.xlabel("$t (s) \\rightarrow$")
plt.ylabel("$x (units) \\rightarrow$")
plt.show()
```

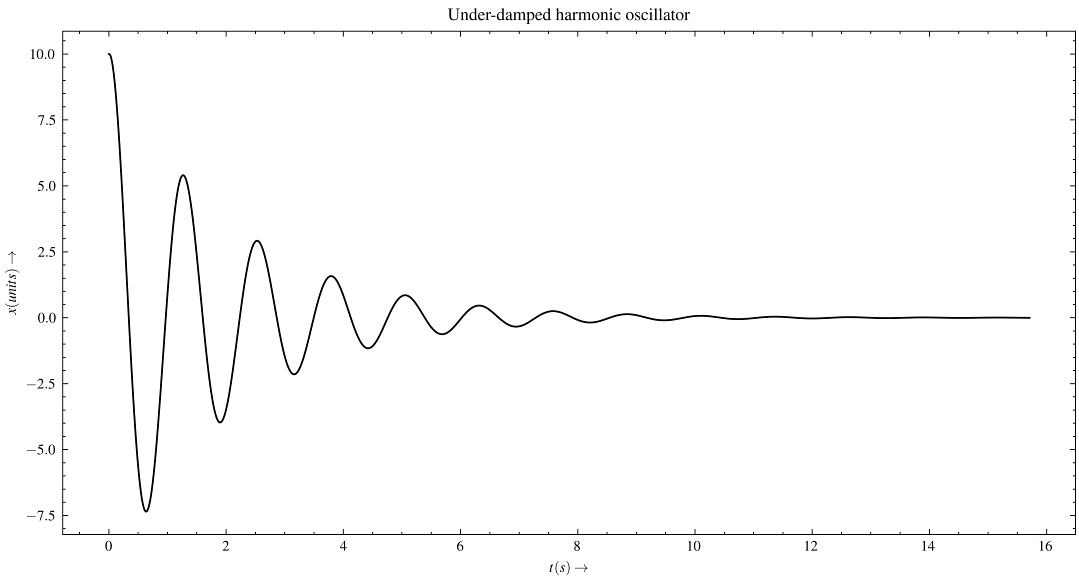


Under-damped harmonic oscillator

$$\gamma^2 < \omega^2$$

[12]:
```python
gamma = 0.5
omega = 5


def x_prime_prime(t, x, x_prime):
    return - 2 * gamma * x_prime - omega**2 * x


t_i = 0
t_f = 5 * pi
x_i = 10
x_prime_i = 1
h = 0.001
```

[13]:
```python
points, x_under = solve_euler(x_prime_prime, t_i, t_f, x_i, x_prime_i, h)

plt.plot(points, x_under)
plt.title("Under-damped harmonic oscillator")
plt.xlabel("$t (s) \\rightarrow$")
plt.ylabel("$x (units) \\rightarrow$")
plt.show()
```

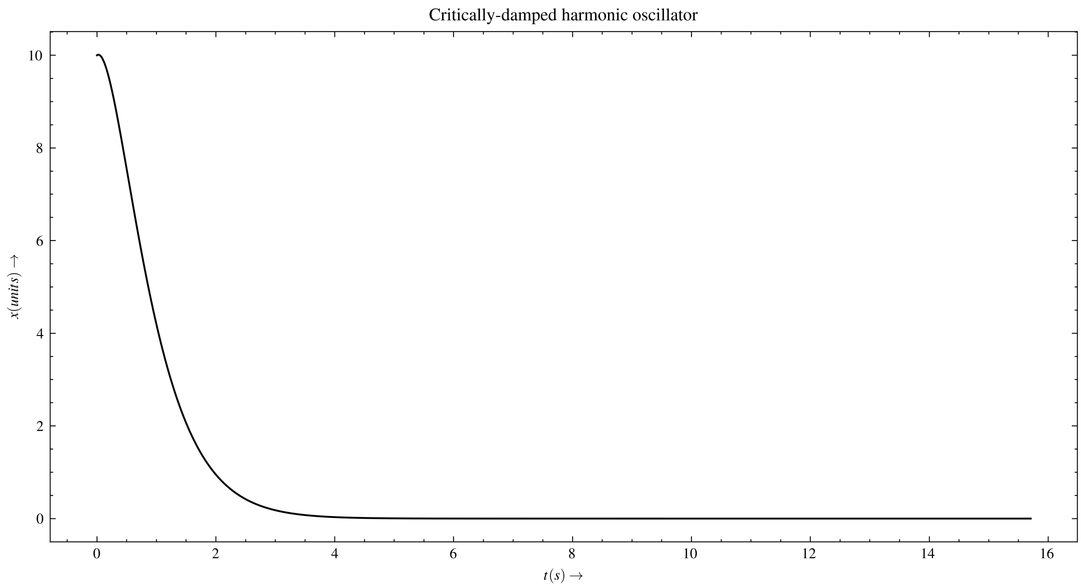


Critically-damped harmonic oscillator

$$\gamma^2 = \omega^2$$

[14]:
```
gamma = 2
omega = 2


def x_prime_prime(t, x, x_prime):
    return - 2 * gamma * x_prime - omega**2 * x


t_i = 0
t_f = 5 * pi
x_i = 10
x_prime_i = 1
h = 0.001
```

[15]:
```
points, x_critical = solve_euler(
    x_prime_prime, t_i, t_f, x_i, x_prime_i, h)

plt.plot(points, x_critical)
plt.title("Critically-damped harmonic oscillator")
plt.xlabel("$t (s) \\rightarrow$")
plt.ylabel("$x (units) \\rightarrow$")
plt.show()
```

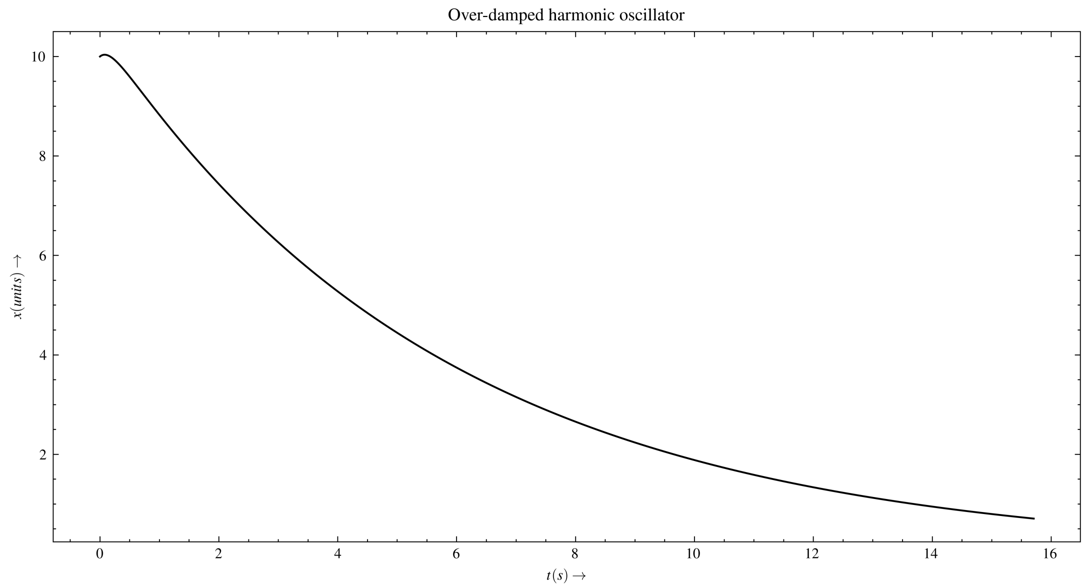


Over-damped harmonic oscillator

$$\gamma^2 > \omega^2$$

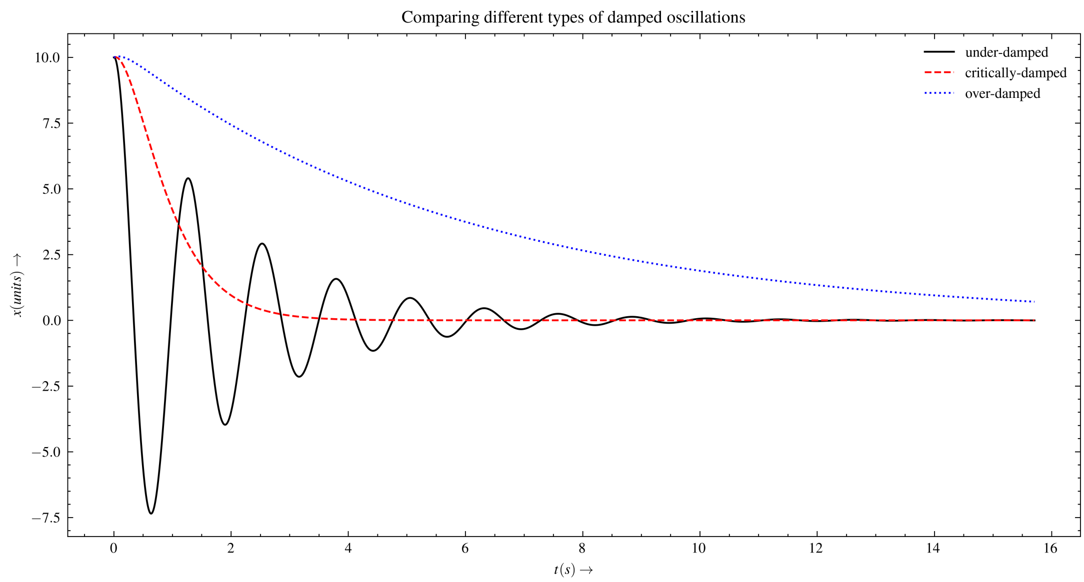
[16]:
```
gamma = 3
omega = 1


def x_prime_prime(t, x, x_prime):
    return - 2 * gamma * x_prime - omega**2 * x


t_i = 0
t_f = 5 * pi
x_i = 10
x_prime_i = 1
h = 0.001
```

[17]:
```
points, x_over = solve_euler(x_prime_prime, t_i, t_f, x_i, x_prime_i, h)

plt.plot(points, x_over)
plt.title("Over-damped harmonic oscillator")
plt.xlabel("$t (s) \\rightarrow$")
plt.ylabel("$x (units) \\rightarrow$")
plt.show()
```



10.3.2 Comparing different types of damped oscillations

[18]:
```
plt.plot(points, x_under, label="under-damped")
plt.plot(points, x_critical, label="critically-damped")
plt.plot(points, x_over, label="over-damped")
plt.title("Comparing different types of damped oscillations")
plt.xlabel("$t (s) \\rightarrow$")
plt.ylabel("$x (units) \\rightarrow$")
plt.legend()
plt.show()
```



Forced harmonic oscillator

For forced harmonic oscillator the quation of motion becomes,

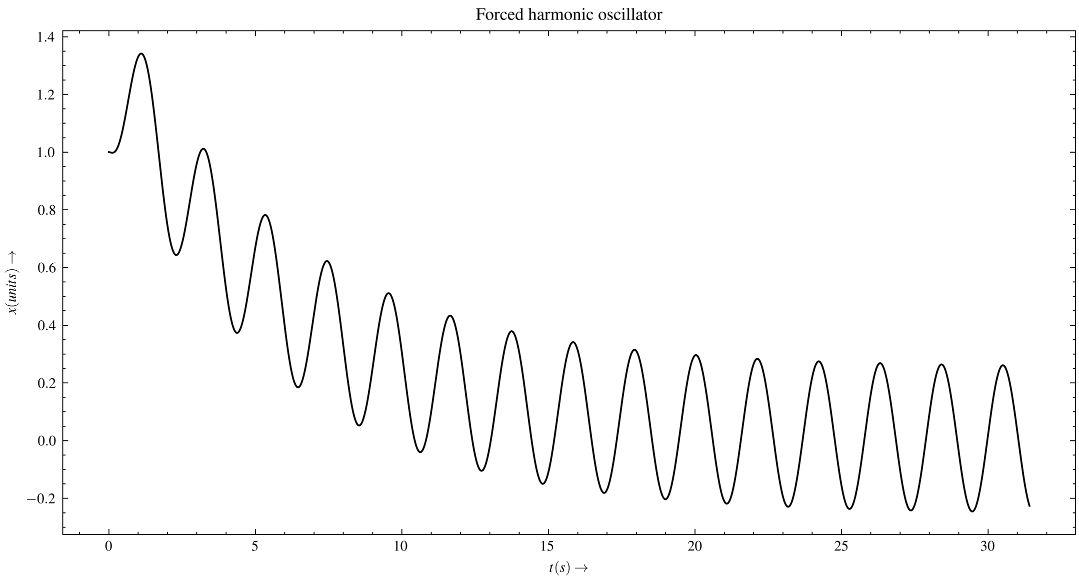
$$\ddot{x} + 2\gamma\dot{x} + \omega^2 x = f_o sin(\alpha t)$$

$$\ddot{x} = -2\gamma\dot{x} + \omega^2 x - f_o sin(\alpha t)$$

where, f_o is the apmlituted and α is the andgular frequency of the given force.

[19]:
```
gamma = 3
omega = 1
f_0 = 5
alpha = 3


def x_prime_prime(t, x, x_prime):
    return - 2 * gamma * x_prime - omega**2 * x + f_0 * sin(alpha * t)


t_i = 0
t_f = 10 * pi
x_i = 1
x_prime_i = 0
h = 0.001
```

[20]:
```
points, x_forced = solve_euler(x_prime_prime, t_i, t_f, x_i, x_prime_i, h)

plt.plot(points, x_forced)
plt.title("Forced harmonic oscillator")
plt.xlabel("$t (s) \\rightarrow$")
plt.ylabel("$x (units) \\rightarrow$")
plt.show()
```

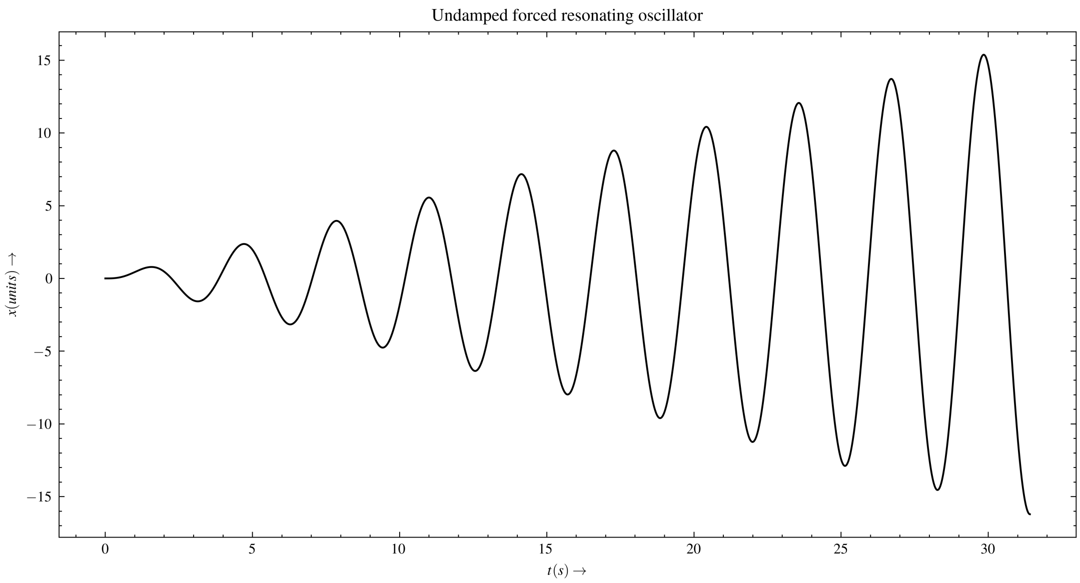


Undamped forced resonating oscillator For forced harmonic oscillator the quation of motion becomes,

$$\ddot{x} + \omega^2 x = f_o sin(\alpha x)$$

[21]:
```
gamma = 0
omega = 2
f_0 = 2
alpha = 2


def x_prime_prime(t, x, x_prime):
    return - 2 * gamma * x_prime - omega**2 * x + f_0 * sin(alpha * t)


t_i = 0
t_f = 10 * pi
x_i = 0
x_prime_i = 0
h = 0.001
```

[22]:
```
points, x_resonance = solve_euler(x_prime_prime, t_i, t_f, x_i, x_prime_i, h)

plt.plot(points, x_resonance)
plt.title("Undamped forced resonating oscillator")
plt.xlabel("$t (s) \\rightarrow$")
plt.ylabel("$x (units) \\rightarrow$")
plt.show()
```



10.3.3 Projectile Motion

A projectile motion is characterised by,

$$\frac{d^2x}{dt^2} = 0$$

$$\frac{d^2y}{dt^2} = -g$$

where, g is acceleration due to gravity.

[23]:
```
def x_prime_prime(t, x, x_prime):
    return 0.0
```

```python
def y_prime_prime(t, y, y_prime):
    return - g


u = 100
thetas = [15, 30, 45, 60]
x_i, y_i = (0, 0)

t_i = 0
t_f = 25
h = 0.001
```

[24]:
```python
# Plotting projectile motion for different angles with same initial velocity
for theta in thetas:

    x_prime_i = u * cos(theta * (pi / 180))
    y_prime_i = u * sin(theta * (pi / 180))

    points, x = solve_euler(x_prime_prime, t_i, t_f, x_i, x_prime_i, h)

    points, y = solve_euler(y_prime_prime, t_i, t_f, y_i, y_prime_i, h)
    y[y < 0] = None

    plt.plot(x, y, label=f"$u = {u}, \\theta = {theta}^\\circ$")

plt.title("Projectile Motion")
plt.xlabel("$x (units) \\rightarrow$")
plt.ylabel("$y (units) \\rightarrow$")
plt.legend()
plt.show()
```