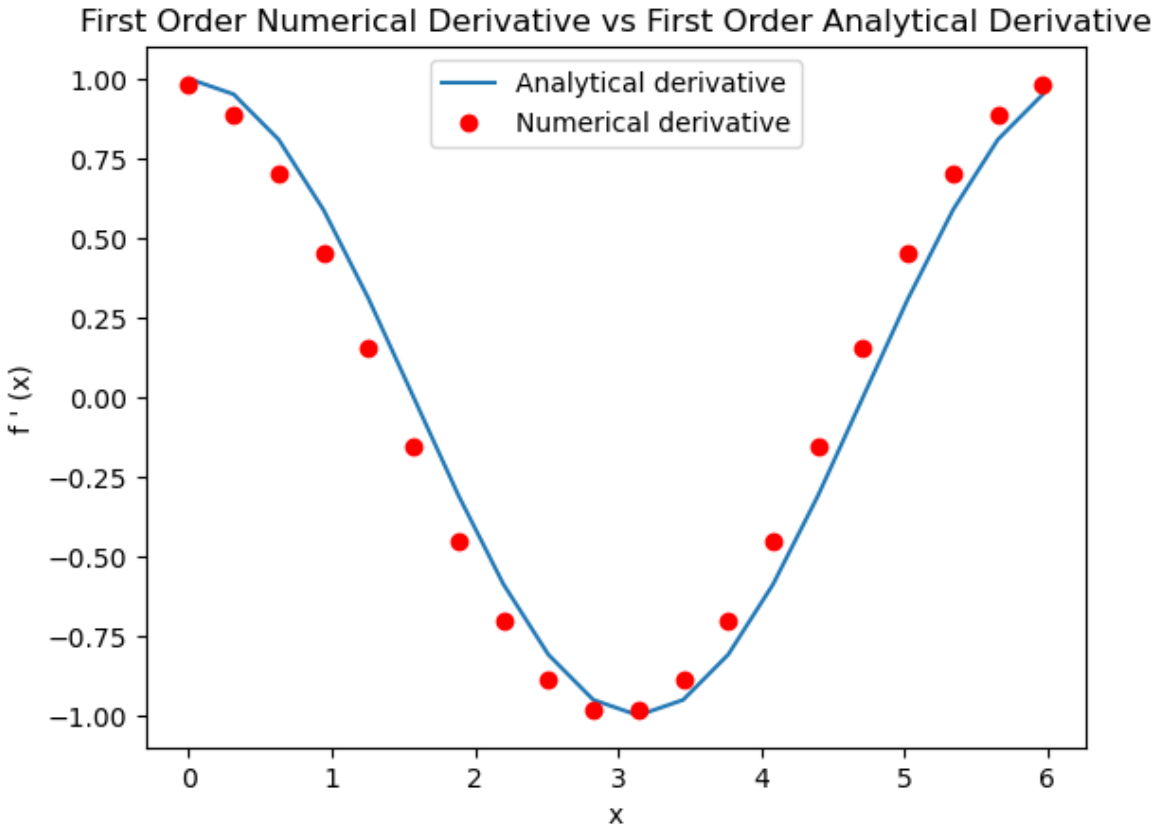
2.1

a)

$$f'(x) \approx \frac{f(x+\Delta x) - f(x)}{\Delta x} \tag{2.2}$$

Using N = 20 gridpoints `np.linspace(0, 2 * np.pi, 20,endpoint=False)` evaluate the numerical derivative of $f(x)$ using the above formula. Plot the result against the true $f'(x)$N

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
N = 20
x = np.linspace(0, 2 * np.pi, N, endpoint=False)
dx = 2 * np.pi / N

fx = np.sin(x)
f_prime_numerical = (np.roll(fx, -1) - fx) / dx
f_prime_analytical = np.cos(x)

plt.plot(x, f_prime_analytical, label='Analytical derivative')
plt.plot(x, f_prime_numerical, 'ro', label='Numerical derivative')
plt.title('First Order Numerical Derivative vs First Order Analytical Der
plt.xlabel('x')
plt.ylabel("f ' (x)")
plt.legend()
plt.show()
```



b)

Make the same plots for the following numerical derivative schemes:

	$-3\Delta x$	$-2\Delta x$	$-\Delta x$	0	Δx	$2\Delta x$	$3\Delta x$
$O(\Delta x)$	0	0	0	-1	1	0	0
$O\left(\Delta x^2\right)$	0	0	$-\frac{1}{2}$	0	$\frac{1}{2}$	0	0
$O\left(\Delta x^4\right)$	0	$\frac{1}{12}$	$-\frac{2}{3}$	0	$\frac{2}{3}$	$-\frac{1}{12}$	0
$O\left(\Delta x^6\right)$	$-\frac{1}{60}$	$\frac{3}{20}$	$-\frac{3}{4}$	0	$\frac{3}{4}$	$-\frac{3}{20}$	$\frac{1}{60}$

Using the table we can get the following formulas:

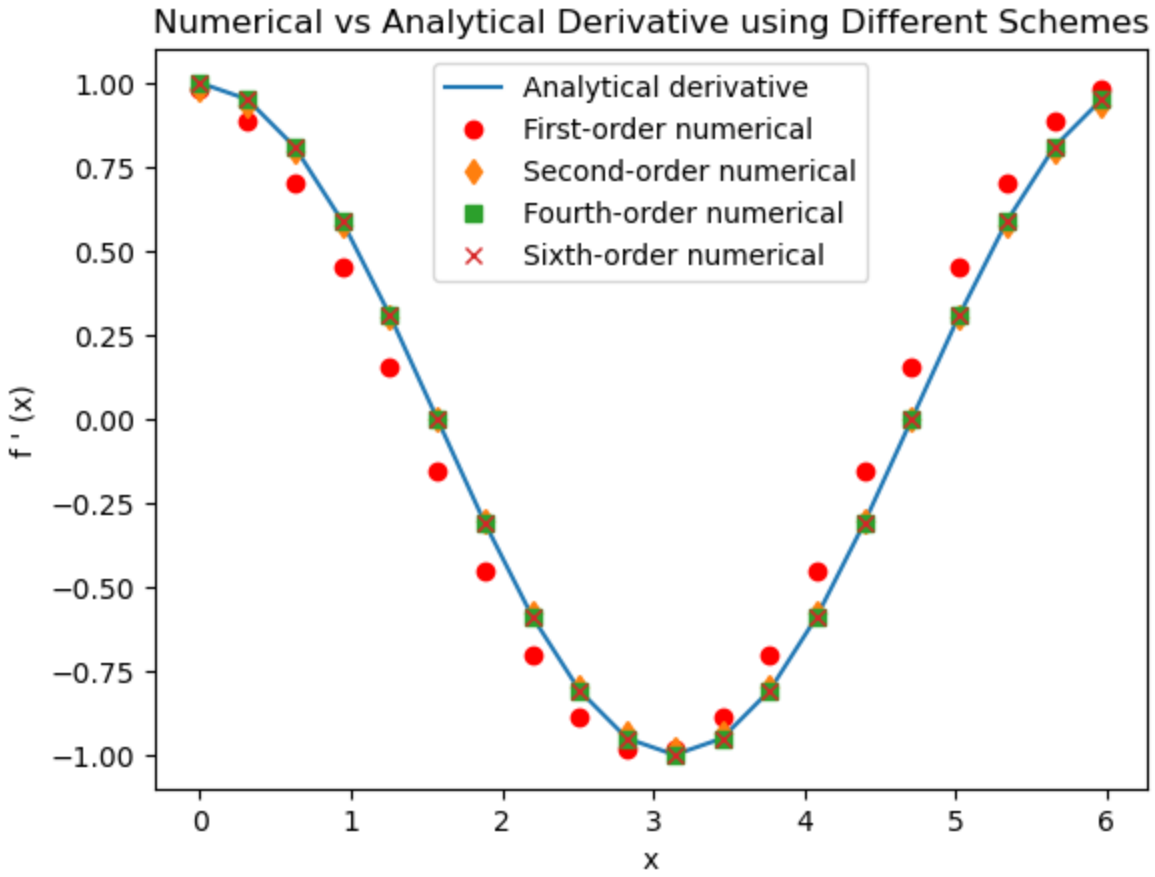
second order = $f(x - \Delta x) - f(x + \Delta x)/2\Delta x$

fourth order = $f(x + 2\Delta x) - f(x - 2\Delta x) + 8f(x - \Delta x) - 8f(x + \Delta x)/12\Delta x$

sixth order=

$$f(x - 3\Delta x) - f(x + 3\Delta x) + 9f(x + 2\Delta x) - \quad (1)$$
$$9f(x - 2\Delta x) + 45f(x - \Delta x) - 45f(x + \Delta x)\,/60\Delta x \quad (2)$$

In [ ]:
```
def second_order(fx, dx):
    return (np.roll(fx, -1) - np.roll(fx, 1)) / (2 * dx)

def fourth_order(fx, dx):
    return (np.roll(fx, 2) - np.roll(fx, -2) +
            8 * np.roll(fx, -1) - 8 * np.roll(fx, 1)) / (12 * dx)

def sixth_order(fx, dx):
    return (np.roll(fx, -3) - np.roll(fx, 3) +
            9 * np.roll(fx, 2) - 9 * np.roll(fx, -2) +
            45 * np.roll(fx, -1) - 45 * np.roll(fx, 1)) / (60 * dx)

f_prime_second_order = second_order(fx, dx)
f_prime_fourth_order = fourth_order(fx, dx)
f_prime_sixth_order = sixth_order(fx, dx)

plt.plot(x, f_prime_analytical, label='Analytical derivative')
plt.plot(x, f_prime_numerical, 'ro', label='First-order numerical')
plt.plot(x, f_prime_second_order, 'd', label='Second-order numerical')
plt.plot(x, f_prime_fourth_order, 's', label='Fourth-order numerical')
plt.plot(x, f_prime_sixth_order, 'x', label='Sixth-order numerical')
plt.xlabel('x')
plt.ylabel("f ' (x)")
plt.legend()
plt.title('Numerical vs Analytical Derivative using Different Schemes')
plt.show()
```



c)

Evaluate the maximum absolute error made by the methods above over the entire domain $[0, 2\pi)$

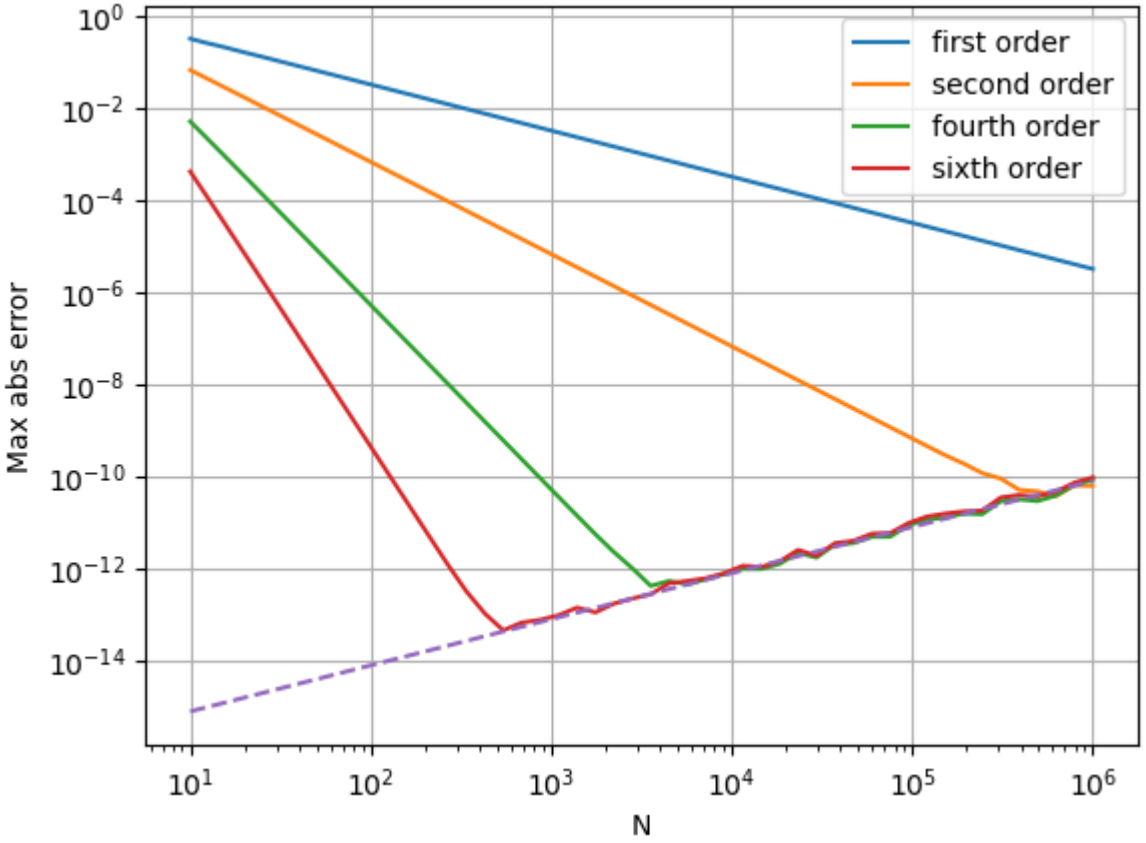
In [ ]:
```
max_error_first_order = np.max(np.abs(f_prime_numerical -
                                      f_prime_analytical))
max_error_second_order = np.max(np.abs(f_prime_second_order -
                                       f_prime_analytical))
max_error_fourth_order = np.max(np.abs(f_prime_fourth_order -
                                       f_prime_analytical))
max_error_sixth_order = np.max(np.abs(f_prime_sixth_order -
                                      f_prime_analytical))

print(f"Max error first-order: {max_error_first_order}")
print(f"Max error second-order: {max_error_second_order}")
print(f"Max error fourth-order: {max_error_fourth_order}")
print(f"Max error sixth-order: {max_error_sixth_order}")
```

```
Max error first-order: 0.15579194727527879
Max error second-order: 0.016368356916534044
Max error fourth-order: 0.0003209038182584445
Max error sixth-order: 6.736514570726548e-06
```

d)

Make a loglog plot of the maximum absolute error of each method as a function of $N \in [10, 106]$. Explain the plot

In [ ]:
```
import numpy as np
N = np.logspace(1, 6, 50, dtype=int)
errors_first_order = []
errors_second_order = []
errors_fourth_order = []
errors_sixth_order = []

for n in N:
    x = np.linspace(0, 2 * np.pi, n, endpoint=False)
    dx = 2 * np.pi / n
    fx = np.sin(x)
    f_prime_analytical = np.cos(x)
    f_prime_numerical = (np.roll(fx, -1) - fx) / dx

    f_prime_second_order = second_order(fx, dx)
    f_prime_fourth_order = fourth_order(fx, dx)
    f_prime_sixth_order = sixth_order(fx, dx)
    max_error_first_order = np.max(np.abs(f_prime_numerical -
                                          f_prime_analytical))
    max_error_second_order = np.max(np.abs(f_prime_second_order -
                                           f_prime_analytical))
    max_error_fourth_order = np.max(np.abs(f_prime_fourth_order -
                                           f_prime_analytical))
    max_error_sixth_order = np.max(np.abs(f_prime_sixth_order -
                                          f_prime_analytical))
    errors_first_order += [max_error_first_order]
    errors_second_order += [max_error_second_order]
    errors_fourth_order += [max_error_fourth_order]
    errors_sixth_order += [max_error_sixth_order]

plt.loglog(N, errors_first_order, label = 'first order')
plt.loglog(N, errors_second_order, label = 'second order')
plt.loglog(N, errors_fourth_order, label = 'fourth order')
plt.loglog(N, errors_sixth_order, label = 'sixth order')
plt.loglog(N, 5*10**(-16)/(2 * np.pi / N), '--')
plt.xlabel('N')
plt.grid(True)
plt.ylabel('Max abs error')
plt.legend()
plt.show()
```



The plot shows the max absolute error of the different order schemes. The max absolute error is plotted along the number of grid points. We see that all the n-order schemes follow a linear line after they reach their lowest value of max absolute error. Also we see that higher order schemes take less grid points to reach their lowest value of max absolute error. When the max absolute error is high, accuracy is low and when the max absolute error is low, accucacy is high.

e)

What is the best accuracy that you obtain with the second order method? and with the sixth order method? can you predict the best accuracy for the first order method without evaluating higher N?

The best accuracy for the second and sixth order is where they have the lowest y value or where max abs error is the lowest.

second order is around 10^{-10}

sixth order is around 10^{-13}

As can be seen from the linear dotted line, there is a bottleneck for the accucarcy (low max abs error is high accrucary), each method have some max accuracy, they will never go under the dotted line (they do on the plot beacause of floating point problems). There is essentially a linear line, we can see that the first order line will intersect the line around 10^{-7}. So therefore for the first order the best accucarcy is around 10^{-7}

f)

How many grid points are needed for a second order method to obtain the same accuracy as a fourth order method with $N = 100$ points?

For the fourth order we get an accucarcy of around 10^{-6} at 100 (10^2) grid points.

The second order method reaches the same accrucary at around $10^{3.5}$ gridpoints

g)

Compare the curves to a plot of $\sim \Delta x^n$ where n is the order of the method as stated in the above table. Also plot $5 \cdot 10^{-16}/\Delta x$ and compare it to the best error of the methods

We see that higher order schemes have a lower error for the same number of points and that less grid points are needed to reach best accuracy for higher order schemes.

We see that the accuracy of the numerical schemes follow the line $f(x) = 5 \cdot 10^{-16}/\Delta x$ after it reaches the best accucarcy .

2.3

Consider the following function

$$f(x) = \begin{cases} e^{-x} + ax - 1 & x < 0 \\ x^2 & x > 0 \end{cases} \tag{2.4}$$

which has the derivative

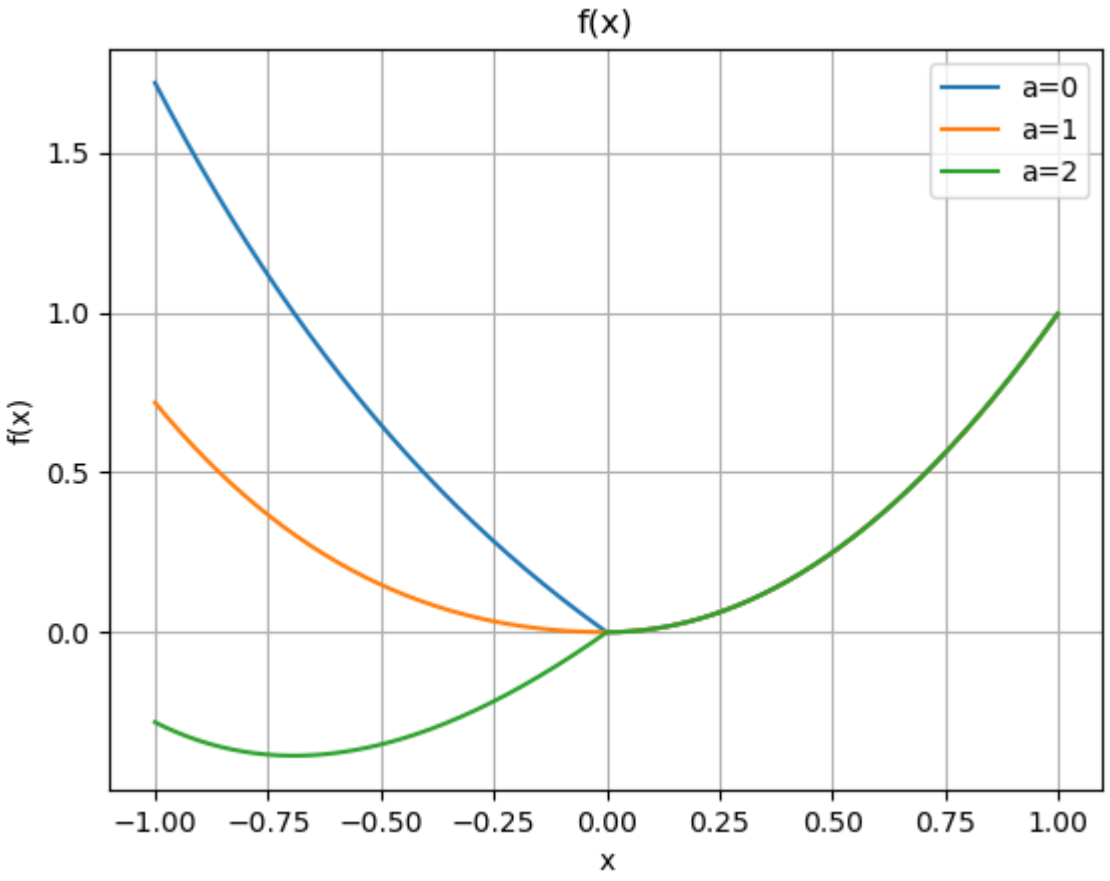
$$f'(x) = \begin{cases} -e^{-x} + a & x < 0 \\ 2x & x > 0 \end{cases} \tag{2.5}$$

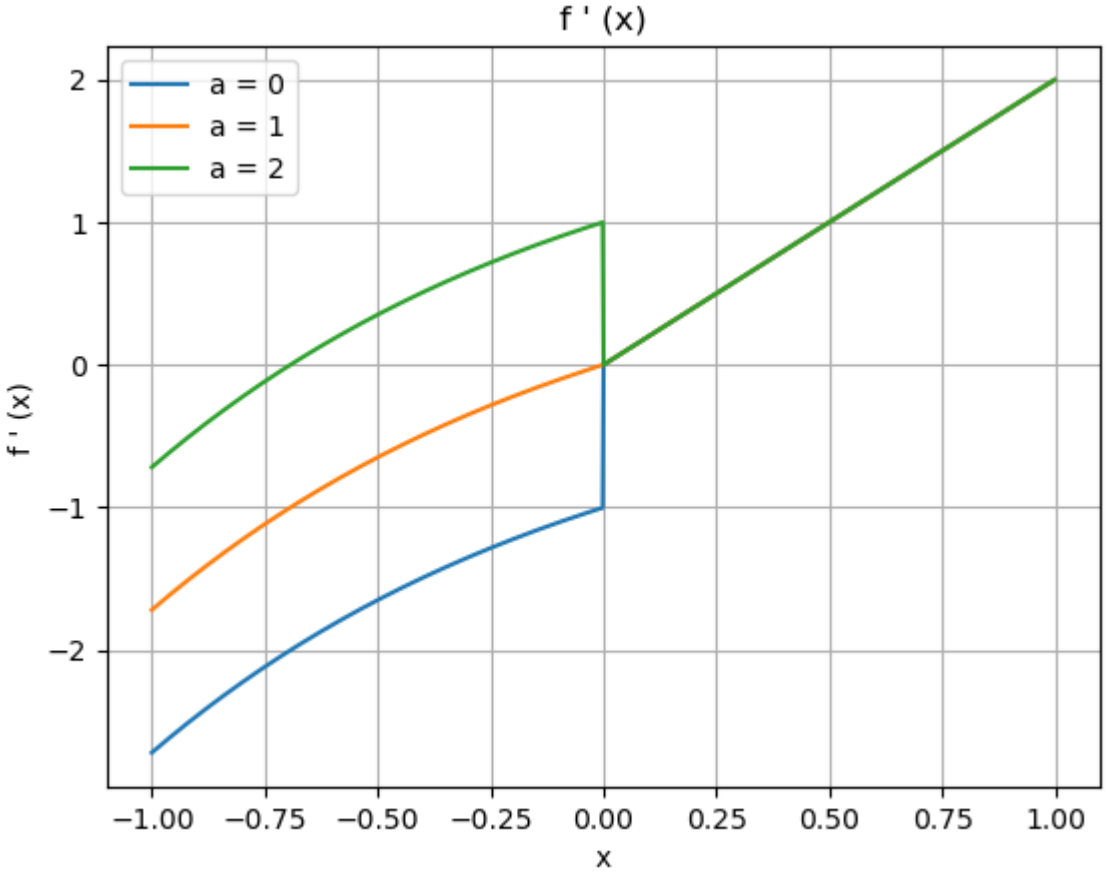
a)

Plot the function and its derivative for $a = 0, a = 1, a = 2$

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

N = 1000
x = np.linspace(-1, 1, N, endpoint=False)

def f(x, a):
    return np.piecewise(x, [x < 0, x > 0],
                        [lambda x: np.exp(-x) + a * x - 1,
                         lambda x: x**2])
```

```
def f_prime(x, a):
    return np.piecewise(x, [x < 0, x > 0],
                        [lambda x: -np.exp(-x) + a,
                         lambda x: 2 * x])

a_values = [0, 1, 2]

# plotting f(x) for different values of a
for a in a_values:
    f_values = f(x, a)
    plt.plot(x, f_values, label=f'a={a}')
    plt.title('f(x)')
    plt.xlabel('x')
    plt.ylabel('f(x)')
    plt.grid(True)
    plt.legend()
plt.show()

# plotting f'(x) for different values of a
for a in a_values:
    f_prime_values = f_prime(x, a)
    plt.plot(x, f_prime_values, label=f'a = {a}')
    plt.title("f ' (x)")
    plt.xlabel('x')
    plt.ylabel("f ' (x)")
    plt.grid(True)
    plt.legend()
plt.show()
```





b)

For what values of $a \in \mathbb{R}$ is $f(0)$ well-defined? i.e. for what values of a do $\lim_{x\to 0^-} f(x) = \lim_{x\to 0^+} f(x)$? What about $f'(0)$?

First lets look at f(0)

for f(0) to be well defined $\lim_{x\to 0^-} f(x)$ have to be equal to $\lim_{x\to 0^+} f(x)$

let first look at $\lim_{x\to 0^-} f(x)$:

$$\lim_{x\to 0^-} f(e^{-x} + ax - 1) = 1 - 1 = 0$$

ax=0 therefore we know that $\lim_{x\to 0^-} f(e^{-x} + ax - 1) = 1 - 1 = 0$ is true for any $a \in \mathbb{R}$

lets look at $\lim_{x\to 0^+} f(x)$:

$$\lim_{x\to 0^+} x^2 = 0$$

this is again true for any $a \in \mathbb{R}$

Therefore since

$$\lim_{x\to 0^-} f(x) = \lim_{x\to 0^+} f(x)$$

$f(0)$ is well defined for any $a \in \mathbb{R}$

Lets look at f'(0):

for f'(0) to be well defined $\lim_{x\to 0^-} f'(x)$ have to be equal to $\lim_{x\to 0^+} f'(x)$

let first look at $\lim_{x\to 0^-} f'(x)$:

$$\lim_{x\to 0^-} (-e^{-x} + a) = -1 + a$$

lets look at $\lim_{x\to 0^+} f'(x)$:

$$\lim_{x\to 0^+} 2x = 0$$

therefore for f'(0) to be well defined: $-1 + a = 0 \implies a = 1$

since $\lim_{x\to 0^-}(-e^{-x} + a) = -1 + a = -1 + 1 = 0$ when $a = 1$

Therefore when a = 1:

$\lim_{x\to 0^-}(-e^{-x} + a) = 0$ and thereby we get:

$\lim_{x\to 0^-} f'(x) = \lim_{x\to 0^+} f'(x)$

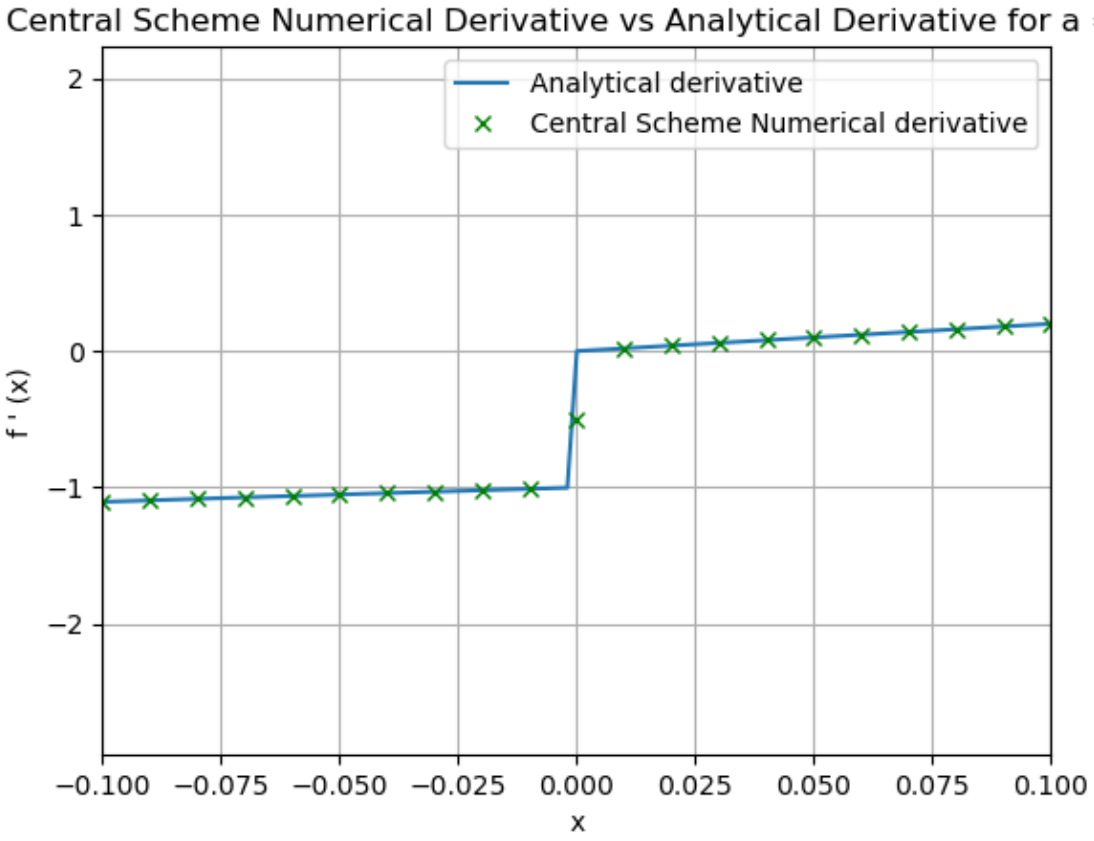
So $f(0)$ is well defined for all $a \in \mathbb{R}$ and $f'(0)$ is well defined for $a = 1$

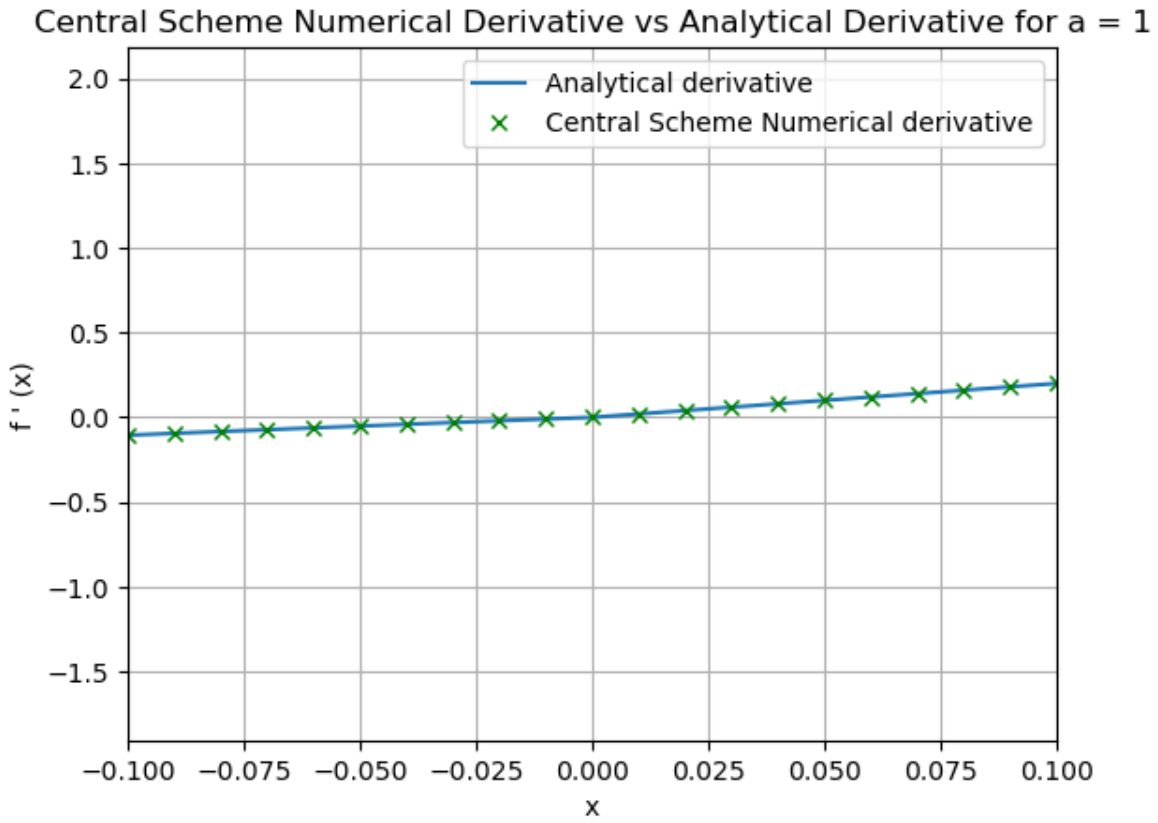
c)

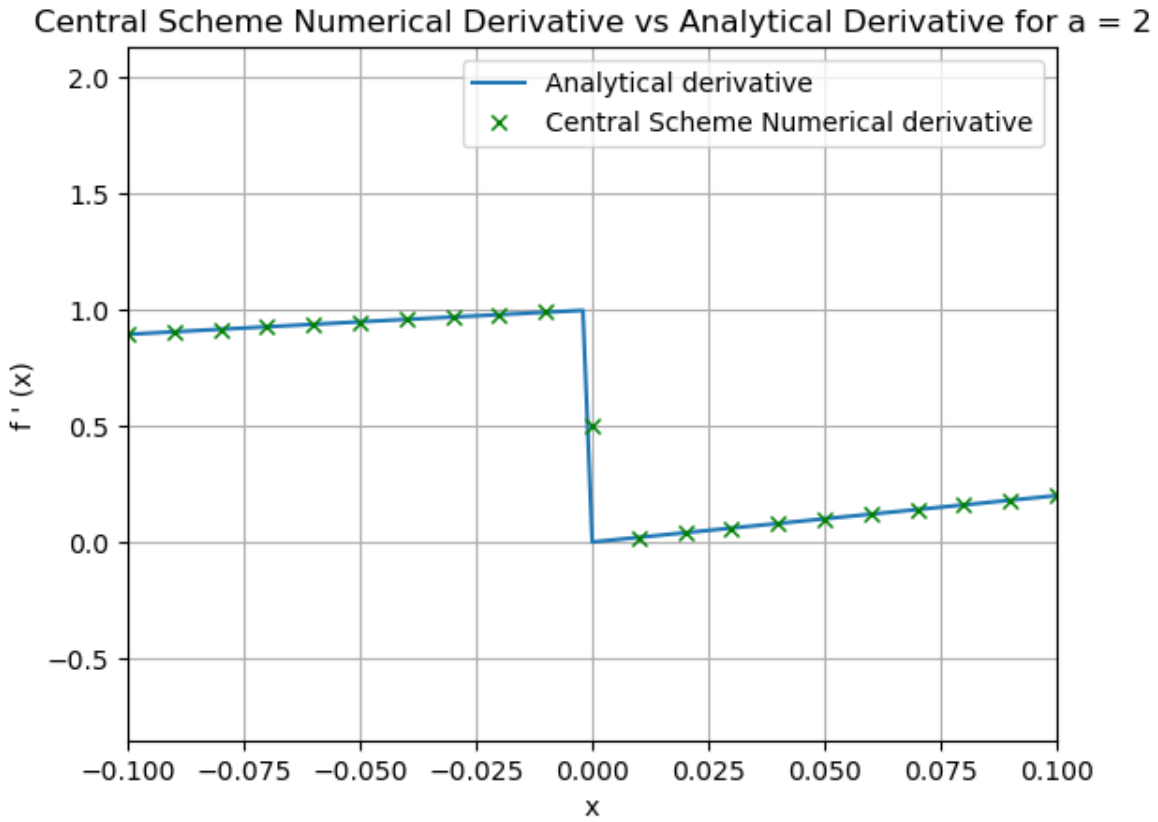
Using the central second order finite difference scheme $\left[O\left(\Delta x^2\right)\right]$, calculate the numerical derivative of $f(x)$ and compare to the analytical $f'(x)$ for $a = 0, a = 1, a = 2$. Plot your results near $x = 0$ and discuss for what values of a you find a good approximation.

In [ ]:
```
dx = x[1] - x[0]

# making the central second order finite difference scheme
A = np.zeros((N, N))

# forward difference
A[0, 0] = -3 / 2
A[0, 1] = 2
A[0, 2] = -1 / 2

# central difference, shifting -1 and 1 to the right for each row
for i in range(1, N - 1):
    A[i, i - 1] = -1/2
    A[i, i + 1] = 1/2

# backward difference
A[N - 1, N - 3] = 1 / 2
A[N - 1, N - 2] = -2
A[N - 1, N - 1] = 3 / 2

for a in a_values:
    f_prime_analytical = f_prime(x, a)
    f_prime_central_scheme = A @ f(x, a) / dx
```

```
    plt.plot(x, f_prime_analytical, label='Analytical derivative')
    plt.plot(x[::5], f_prime_central_scheme[::5],'gx', label='Central Sch
    plt.xlabel('x')
    plt.ylabel("f ' (x)")
    plt.legend()
    plt.grid(True)
    plt.title(f'Central Scheme Numerical Derivative vs Analytical Derivat
    plt.xlim(-0.1, 0.1)  # Zoom in near x=0
    plt.show()

print(A)
```








```
[[-1.5  2.  -0.5 ...  0.   0.   0. ]
 [-0.5  0.   0.5 ...  0.   0.   0. ]
 [ 0.  -0.5  0.  ...  0.   0.   0. ]
 ...
 [ 0.   0.   0.  ...  0.   0.5  0. ]
 [ 0.   0.   0.  ... -0.5  0.   0.5]
 [ 0.   0.   0.  ...  0.5 -2.   1.5]]
```

Looking at the plot, the values from the central scheme numerical derrivivate follows

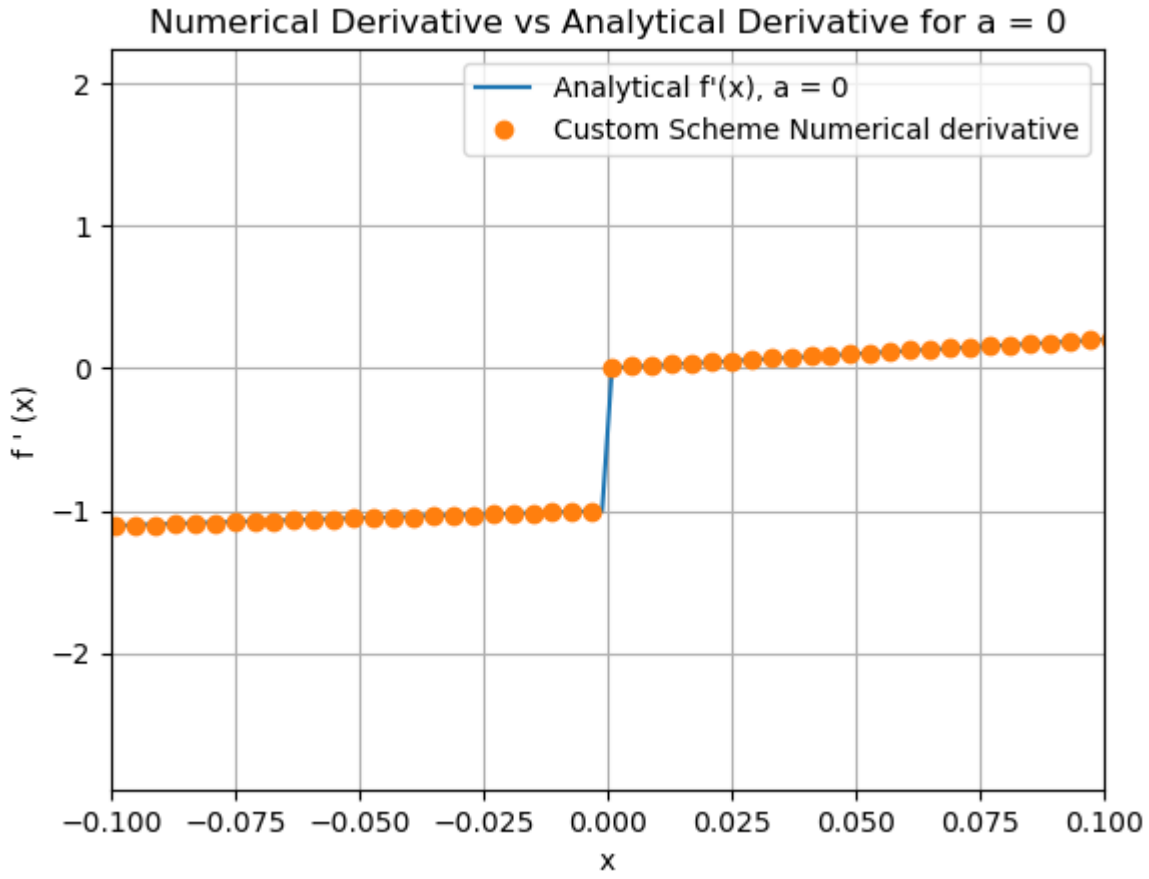
the analytical values the closes when $a = 1$. when a = 0 or a = 2 the central scheme has trouble getting the correct value at x = 0. This makes sense since f'(0) is only well defined for when a = 1, we that know from 2.3 b.

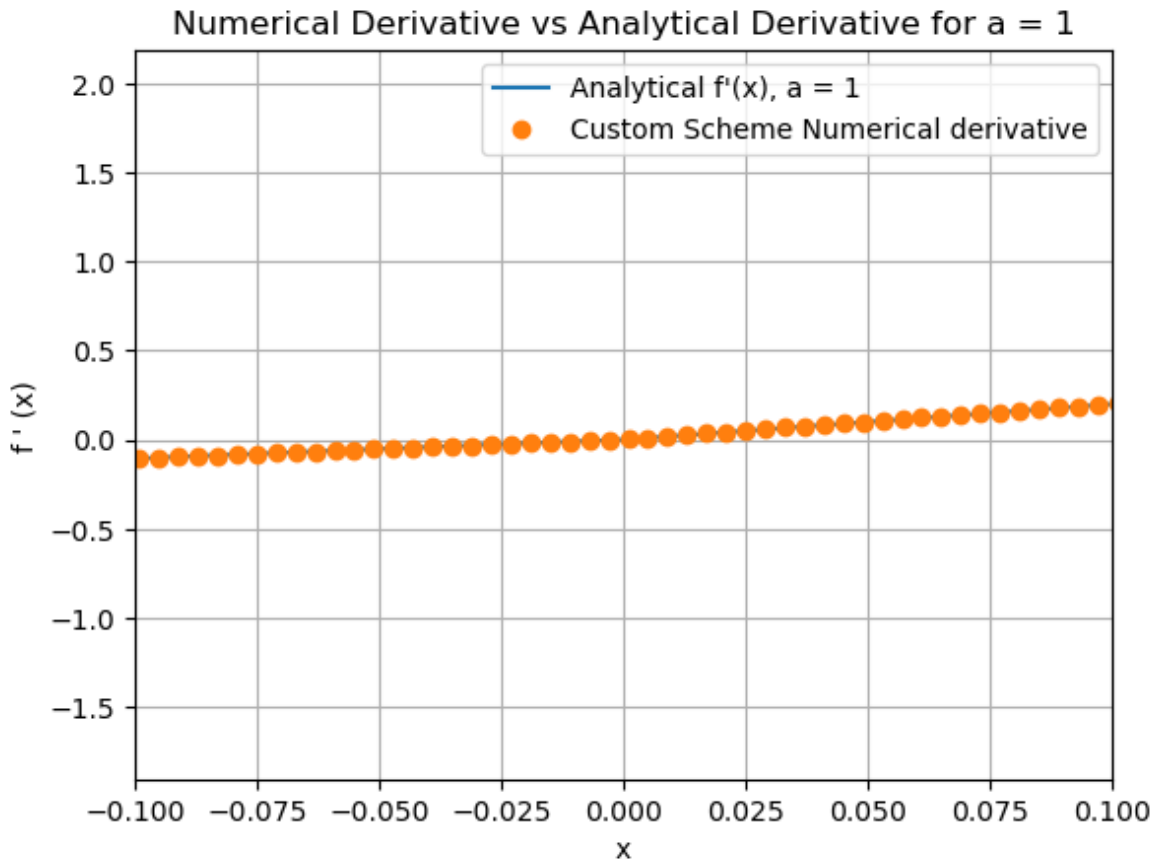
d)

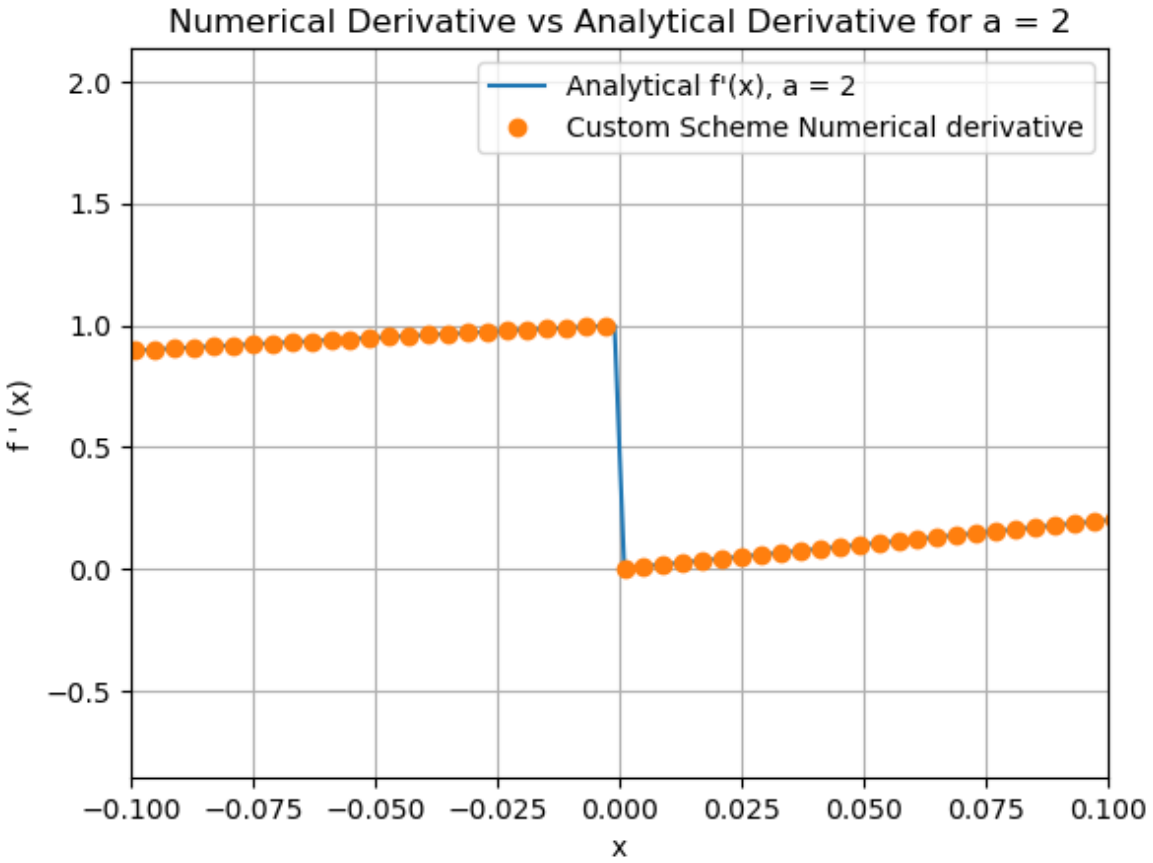
Derive a suitable finite difference scheme for calculating the first derivative of the function in such a way that points from $x < 0$ are never used together with points $x > 0$.

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

# A_ is the finite difference scheme
A_ = np.zeros((N, N))

# Backward difference for x < 0
for i in range(1, N//2): # from start to middle
    A_[i, i] = 1
    A_[i, i-1] = -1

# Forward difference for x > 0
for i in range(N//2, N-1): # from middle to end
    A_[i, i+1] = 1 #
    A_[i, i] = -1 #

print(f"finite difference scheme\n {A_}")

x = np.linspace(-1, 1, 1000)
a_values = [0, 1, 2]

for a in a_values:
    f_prime_custom_scheme = A_ @ f(x, a) / dx
    analytical_derivative = f_prime(x, a)
    plt.plot(x, analytical_derivative, label=f'Analytical f\'(x), a = {a}
    plt.plot(x[::2], f_prime_custom_scheme[::2],'o', label='Custom Scheme
    plt.title(f'Numerical Derivative vs Analytical Derivative for a = {a}
    plt.xlabel("x")
    plt.ylabel("f ' (x)")
    plt.legend()
    plt.xlim([-0.1, 0.1])  # Zoom in near x=0
    plt.grid(True)
    plt.show()
```

```
finite difference scheme
 [[ 0.  0.  0. ...  0.  0.  0.]
 [-1.  1.  0. ...  0.  0.  0.]
 [ 0. -1.  1. ...  0.  0.  0.]
 ...
 [ 0.  0.  0. ... -1.  1.  0.]
 [ 0.  0.  0. ...  0. -1.  1.]
 [ 0.  0.  0. ...  0.  0.  0.]]
```


Numerical Derivative vs Analytical Derivative for a = 0
Analytical f'(x), a = 0
Custom Scheme Numerical derivative
f ' (x)
x
2
1
0
−1
−2
−0.100
−0.075
−0.050
−0.025
0.000
0.025
0.050
0.075
0.100


Numerical Derivative vs Analytical Derivative for a = 1
Analytical f'(x), a = 1
Custom Scheme Numerical derivative
f ' (x)
x
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−0.100
−0.075
−0.050
−0.025
0.000
0.025
0.050
0.075
0.100

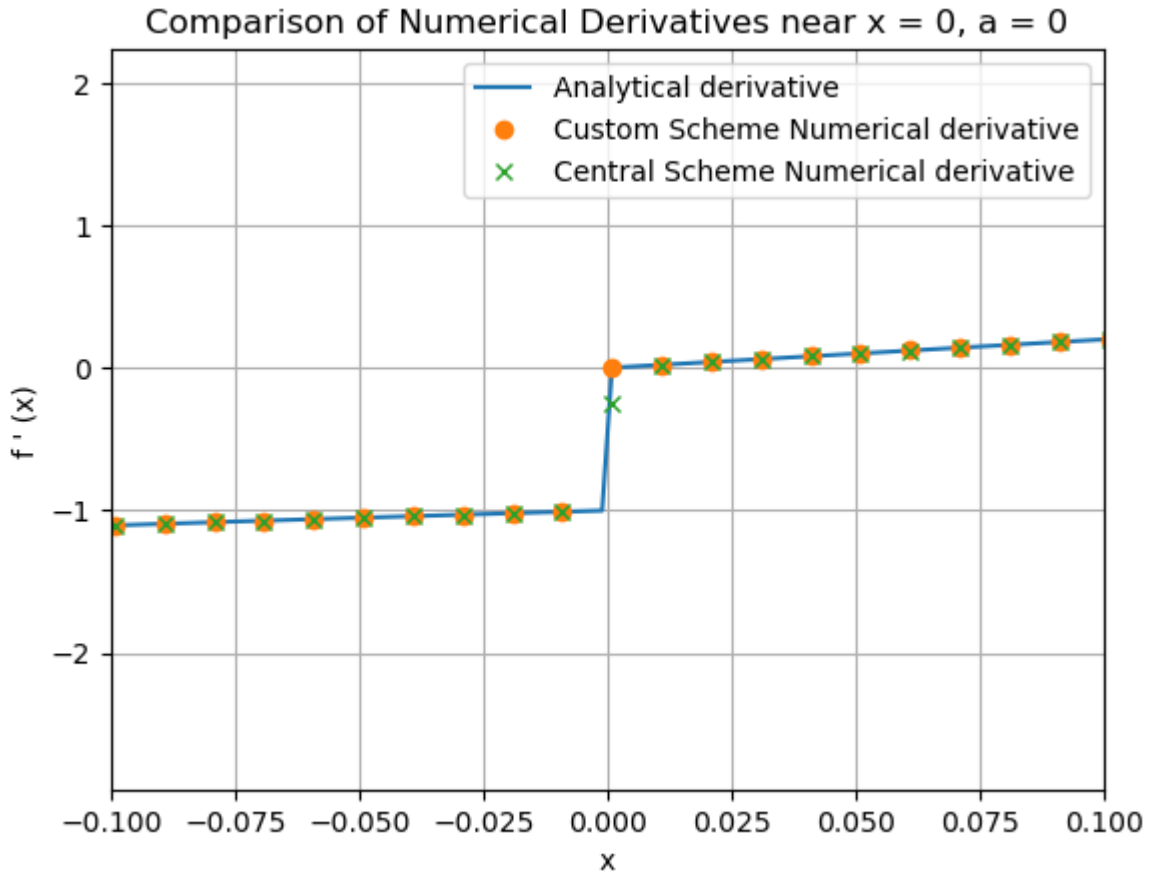


e)

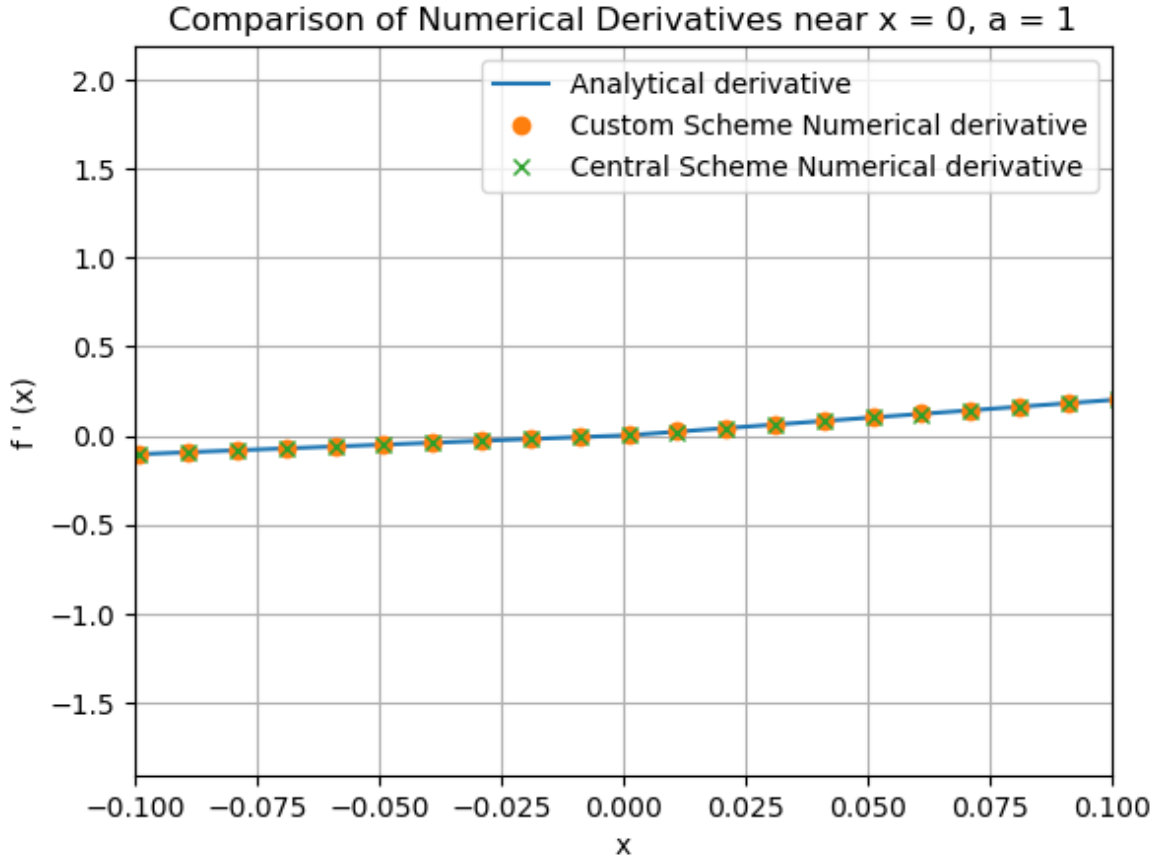
Plot your results near $x = 0$ and compare with the central scheme.

In [ ]:
```
for a in a_values:
    f_prime_custom = A_ @ f(x, a) / dx
    f_prime_central = A @ f(x, a) / dx

    analytical_derivative = f_prime(x, a)

    plt.plot(x, analytical_derivative, label='Analytical derivative')
    plt.plot(x[::5], f_prime_custom[::5],'o', label='Custom Scheme Numeri
    plt.plot(x[::5], f_prime_central[::5],'x', label='Central Scheme Nume
    max_error = np.max(np.abs(f_prime_custom-analytical_derivative))
    print(f"Max error for custom scheme a = {a}: {max_error}")
    max_error = np.max(np.abs(f_prime_central-analytical_derivative))
    print(f"Max error for central scheme a = {a}: {max_error}")
    plt.title(f"Comparison of Numerical Derivatives near x = 0, a = {a}")
    plt.xlabel('x')
    plt.ylabel("f ' (x)")
    plt.xlim(-0.1, 0.1)  # Zoom in near x=0
    plt.legend()
    plt.grid(True)
    plt.show()
```

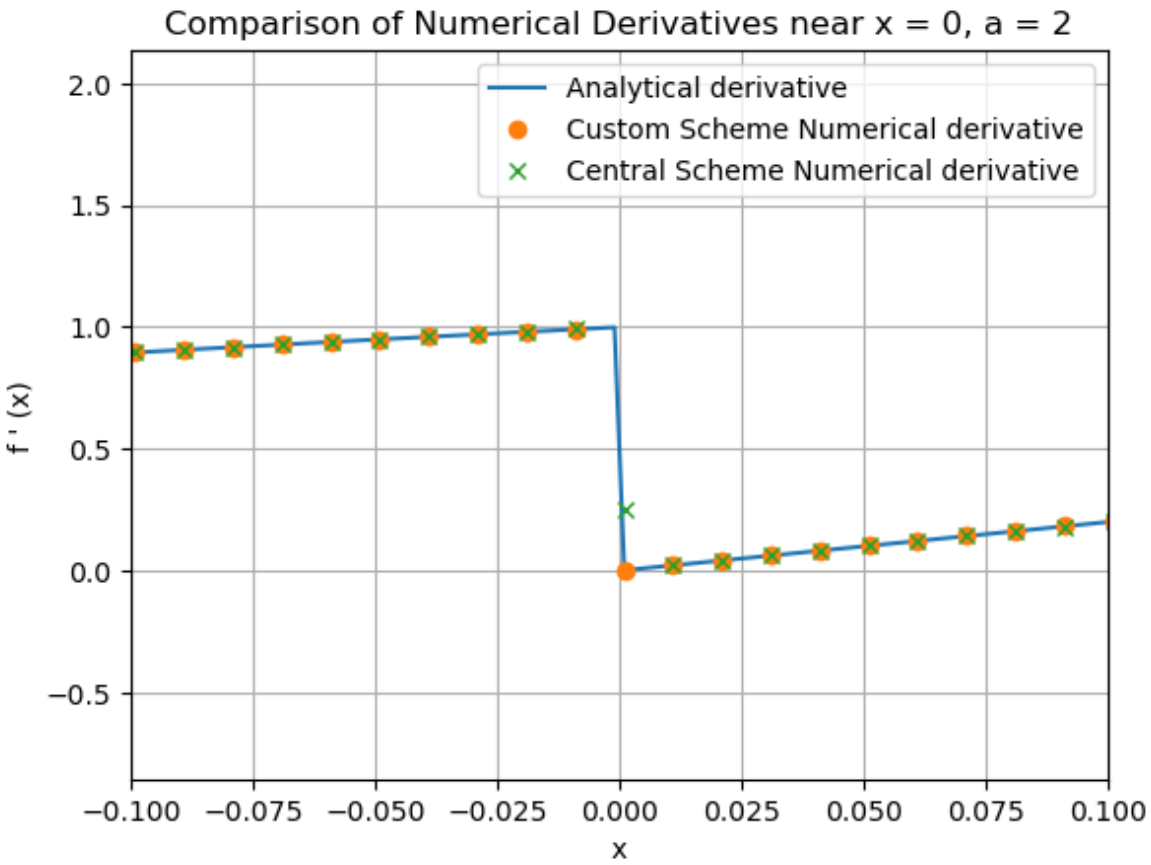
```
Max error for custom scheme a = 0: 2.718281828459045
Max error for central scheme a = 0: 0.250123037671118
```




```
Max error for custom scheme a = 1: 2.0
Max error for central scheme a = 1: 0.0020020020019551055
```




```
Max error for custom scheme a = 2: 2.0
Max error for central scheme a = 2: 0.2503774628293708
```



We see that the max error for the custom scheme is 2.0 which is higher than the central order scheme, meaning that the central scheme has higher accuracy.

3.1

Consider the ODE

$$\frac{\mathrm{d}x}{\mathrm{d}t} = \alpha(\sin t - x) \tag{3.1}$$

For $x(0) = 0$, this has the analytical solution

$$x(t) = \frac{\alpha}{1+\alpha^2}\left(e^{-\alpha t} - \cos t + \alpha \sin t\right) \tag{3.2}$$

a)

Solve the equation using the explicit Euler method and compare with the analytical solution for $\alpha = 0.1$ on $t \in [0, 100]$ using $\Delta t = 0.01$.

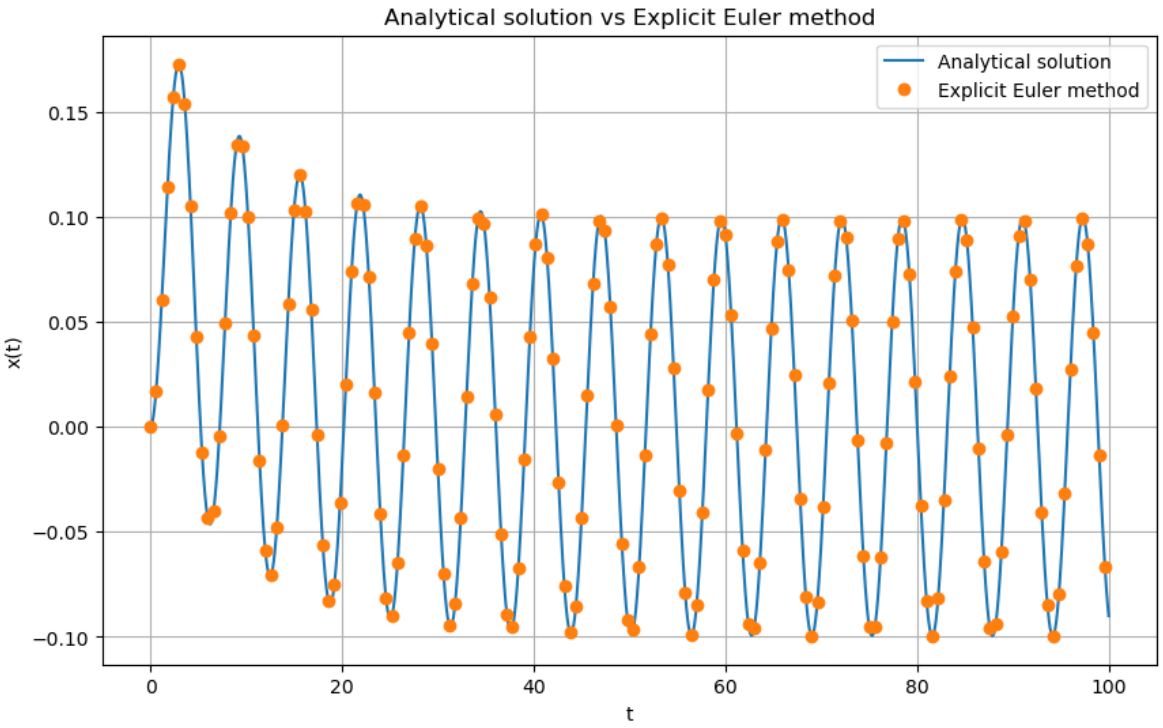
Eulers method:

$$y_{n+1} = y_n + g \cdot f(t_n, y_n)$$

In our case:

$$y_{n+1} = y_n + \Delta t \cdot \alpha(\sin t_n - y_n)$$

In [ ]:
```
alpha = 0.1
dt = 0.01
t = np.arange(0, 100, dt)

def analytical_solution(t):
    global alpha
    return alpha/(1+alpha**2)*(np.exp(-alpha*t)-np.cos(t)+alpha*np.sin(t)

# dx/dt
def f(t, x):
    global alpha
    return alpha * (np.sin(t) - x)

def explicit_euler(f, x0, dt):
    x = np.zeros_like(t)
    x[0] = x0
    for i in range(1, len(t)):
        x[i] = x[i-1] + dt * f(t[i-1], x[i-1])
    return x

x = explicit_euler(f, 0, dt)

plt.figure(figsize=(10, 6))
plt.plot(t, analytical_solution(t), label='Analytical solution')
plt.plot(t[::60], x[::60], "o", label='Explicit Euler method')
plt.xlabel('t')
plt.ylabel('x(t)')
plt.grid(True)
plt.legend()
plt.title('Analytical solution vs Explicit Euler method')
plt.show()
```



b)

Do the same for the implicit Euler method.

Implicit euler formula for dx/dt=f(t,x) is:

$$x_{n+1} = x_n + h \cdot f(t_{n+1}, x_{n+1})$$

in our case:

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

We have a problem, we want to calculate the next value x_{n+1} (which we don't have yet) using x_{n+1}. We would like to use the current value x_n (which we have) to calculate the next value x_{n+1}

x_{n+1} appears on both sides, so lets solve for x_{n+1}

Solving for x_{n+1}:

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

$$x_{n+1} + h \cdot \alpha \cdot x_{n+1} = x_n + h \cdot \alpha \cdot f \sin(t_{n+1})$$

Factoring out x_{n+1}

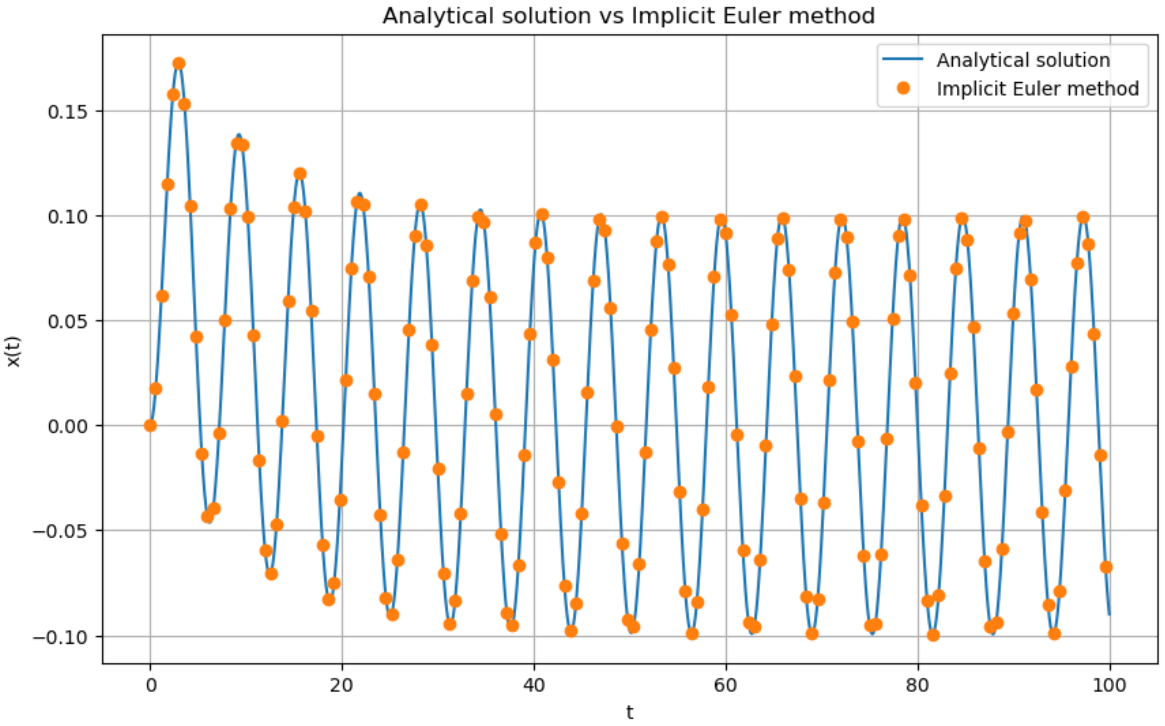
$$x_{n+1} \cdot (1 + h \cdot \alpha) = x_n + h \cdot \alpha \cdot f \sin(t_{n+1})$$

We can now get next value x_{n+1} using only the current value x_n

$$x_{n+1} = \frac{x_n + h \cdot \alpha \cdot f \sin(t_{n+1})}{1 + h \cdot \alpha}$$

In [ ]:
```
def implicit_euler(t, x):
    for i in range(1, len(t)):
        x[i] = (x[i-1] + dt * alpha * np.sin(t[i])) / (1 + dt * alpha)
    return x

# max errors to show that there is a difference between the two methods
# np.equal didn't work, propably because of floating point errors
max_error_explicit_euler = np.max(np.abs(analytical_solution(t) -
                                         explicit_euler(f, 0, dt)))
max_error_implicit_euler = np.max(np.abs(analytical_solution(t) -
                                         implicit_euler(t, x)))
print(f"Max error explicit Euler: {max_error_explicit_euler}")
print(f"Max error implicit Euler: {max_error_implicit_euler}")
print(f"Max erros are not the same: {max_error_explicit_euler !=
                                     max_error_implicit_euler}")

plt.figure(figsize=(10, 6))
plt.plot(t, analytical_solution(t), label='Analytical solution')
plt.plot(t[::60], implicit_euler(t, x)[::60], "o", label='Implicit Euler
plt.xlabel('t')
plt.ylabel('x(t)')
plt.grid(True)
plt.legend()
plt.title('Analytical solution vs Implicit Euler method')
plt.show()
```

```
Max error explicit Euler: 0.0005430522532383492
Max error implicit Euler: 0.0005435657530602905
Max erros are not the same: True
```

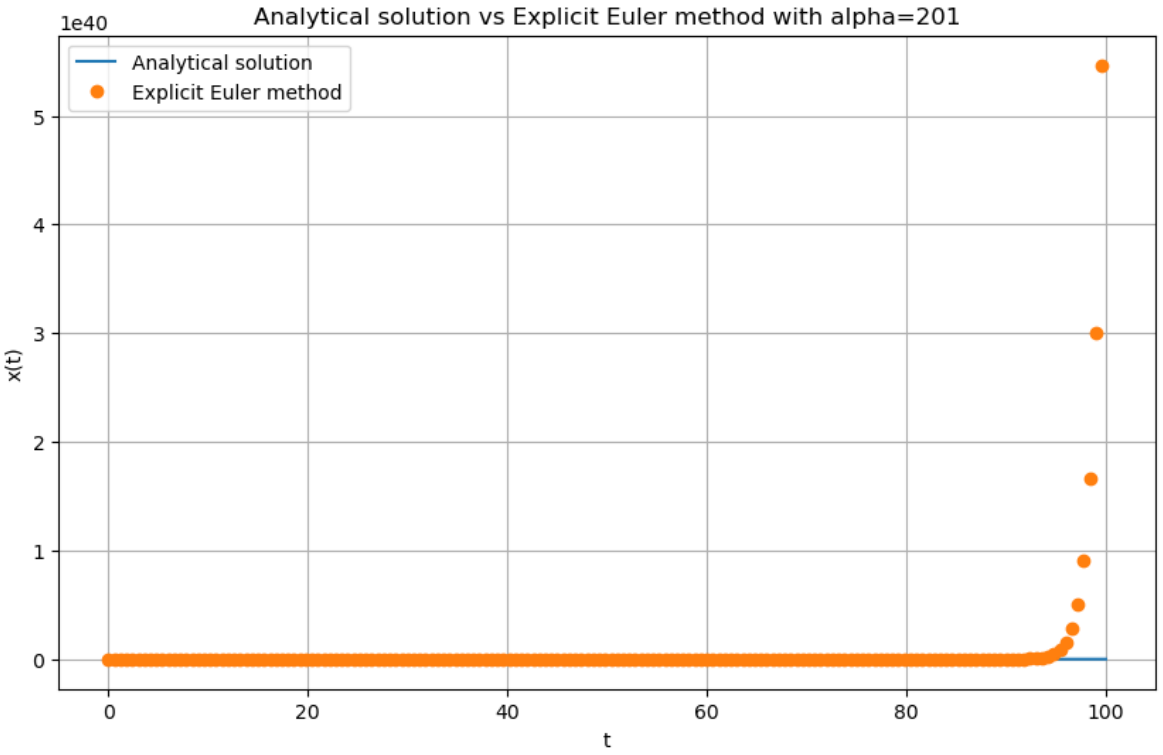


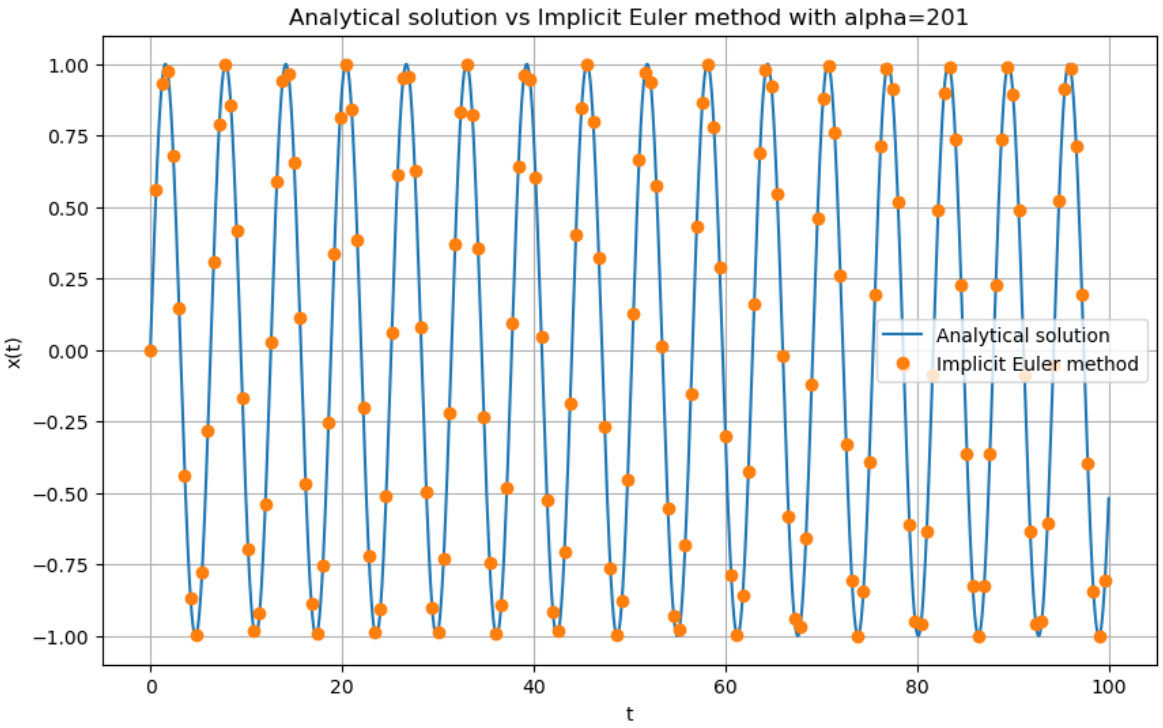
c)

Investigate the behaviour of the two methods for $\alpha > 200$.

In [ ]:
```
alpha = 201
```

```
plt.figure(figsize=(10, 6))
plt.plot(t, analytical_solution(t), label='Analytical solution')
plt.plot(t[::60], explicit_euler(f, 0, dt)[::60], "o", label='Explicit Eu
plt.xlabel('t')
plt.ylabel('x(t)')
plt.grid(True)
plt.legend()
plt.title('Analytical solution vs Explicit Euler method with alpha=201')
plt.show()

plt.figure(figsize=(10, 6))
plt.plot(t, analytical_solution(t), label='Analytical solution')
plt.plot(t[::60], implicit_euler(t, x)[::60], "o", label='Implicit Euler
plt.xlabel('t')
plt.ylabel('x(t)')
plt.grid(True)
plt.legend()
plt.title('Analytical solution vs Implicit Euler method with alpha=201')
plt.show()
```





We see that the explicit euler method performs worse for when $\alpha > 200$ than the implicit euler method. Increasing the time step causes a decrease in accuracy, since we multiply Δx by α, increasing $alpha$ will also increase the time step Δx. The implicit method doesn't have the same problem since we divide by $\alpha \Delta x$ as well as multiply

3.3

Runge-Kutta method

$$
\begin{aligned}
k_1 &= F(f(t), t) \\
k_2 &= F\left(f(t) + \frac{1}{2}k_1\Delta t, t + \frac{1}{2}\Delta t\right) \\
k_3 &= F\left(f(t) + \frac{1}{2}k_2\Delta t, t + \frac{1}{2}\Delta t\right) \\
k_4 &= F\left(f(t) + k_3\Delta t, t + \Delta t\right) \\
f(t + \Delta t) &= f(t) + \frac{1}{6}\left[k_1 + 2k_2 + 2k_3 + k_4\right]\Delta t
\end{aligned}
\tag{3.4}
$$

a)

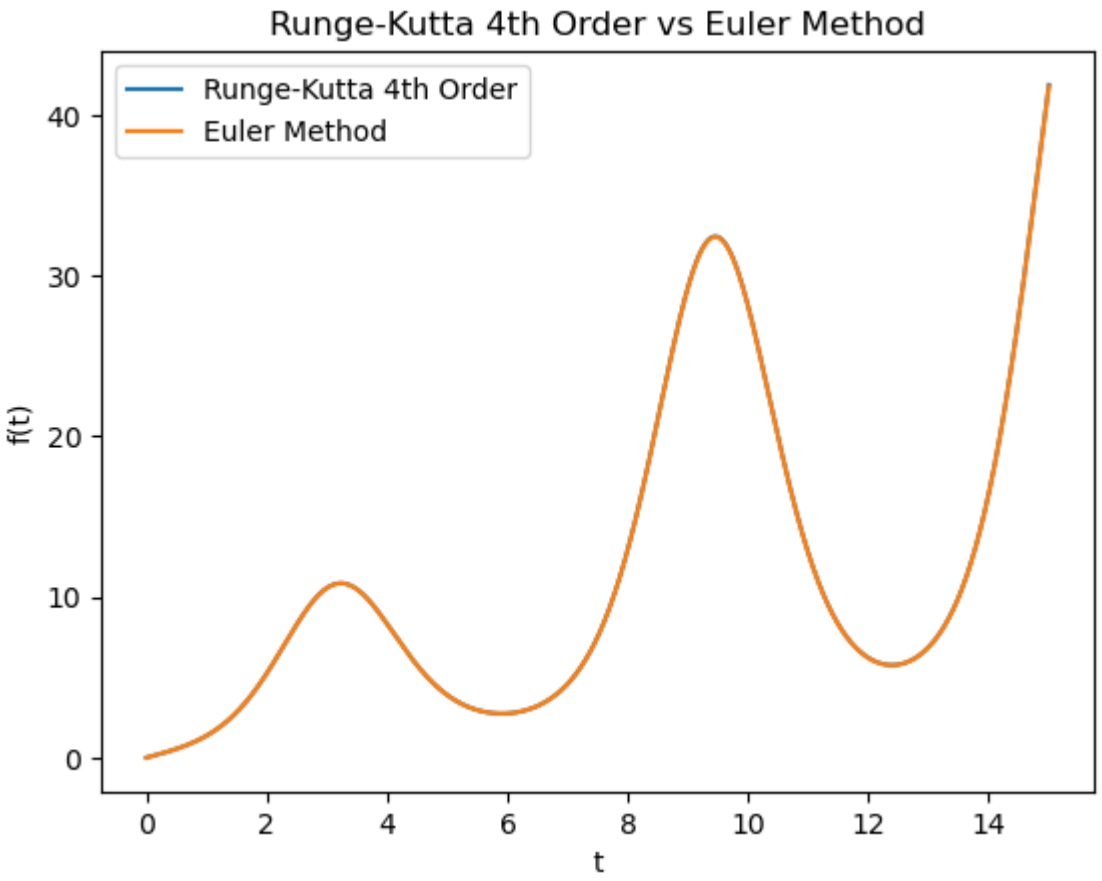
Implement this method to solve

$$f'(t) = 1 + \sin(t) f(t) \tag{3.5}$$

for $t \in [0, 15]$ using $\Delta t = 0.001$ and $f(0) = 0$. Also implement the Euler method for this ODE and check that it gives the same result.

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

```
dt = 0.001

# Define the ODE
def f_prime(f_t, t):
    return 1 + np.sin(t) * f_t

# Runge-Kutta method implementation
def runge_kutta(f_prime, f_0, dt):
    t = np.arange(0, 15, dt)
    f_t = np.zeros_like(t)

    f_values[0] = f_0

    # Iteratively apply Runge-Kutta method
    for i in range(1, len(t)):

        k1 = f_prime(f_t[i-1], t[i-1])
        k2 = f_prime(f_t[i-1] + 0.5 * k1 * dt, t[i-1] + 0.5 * dt)
        k3 = f_prime(f_t[i-1] + 0.5 * k2 * dt, t[i-1] + 0.5 * dt)
        k4 = f_prime(f_t[i-1] + k3 * dt, t[i-1] + dt)

        # Update f(t+Δt)
        f_t[i] = f_t[i-1] + (1/6) * (k1 + 2*k2 + 2*k3 + k4) * dt

    return t, f_t

t_rk, f_rk = runge_kutta(f_prime, 0, dt)

# Euler method
def euler_method(f_prime, x0, dt):
    t = np.arange(0,15, dt)
    x = np.zeros_like(t)
    x[0] = x0

    for i in range(1, len(t)):
        x[i] = x[i-1] + dt * f_prime(x[i-1], t[i-1])

    return t, x


t_euler, x_euler = euler_method(f_prime, 0, dt)

plt.plot(t_rk, f_rk, label='Runge-Kutta 4th Order')
plt.plot(t_euler, x_euler,  label='Euler Method')
plt.xlabel('t')
plt.ylabel('f(t)')
plt.title('Runge-Kutta 4th Order vs Euler Method')
plt.legend()
plt.show()
```

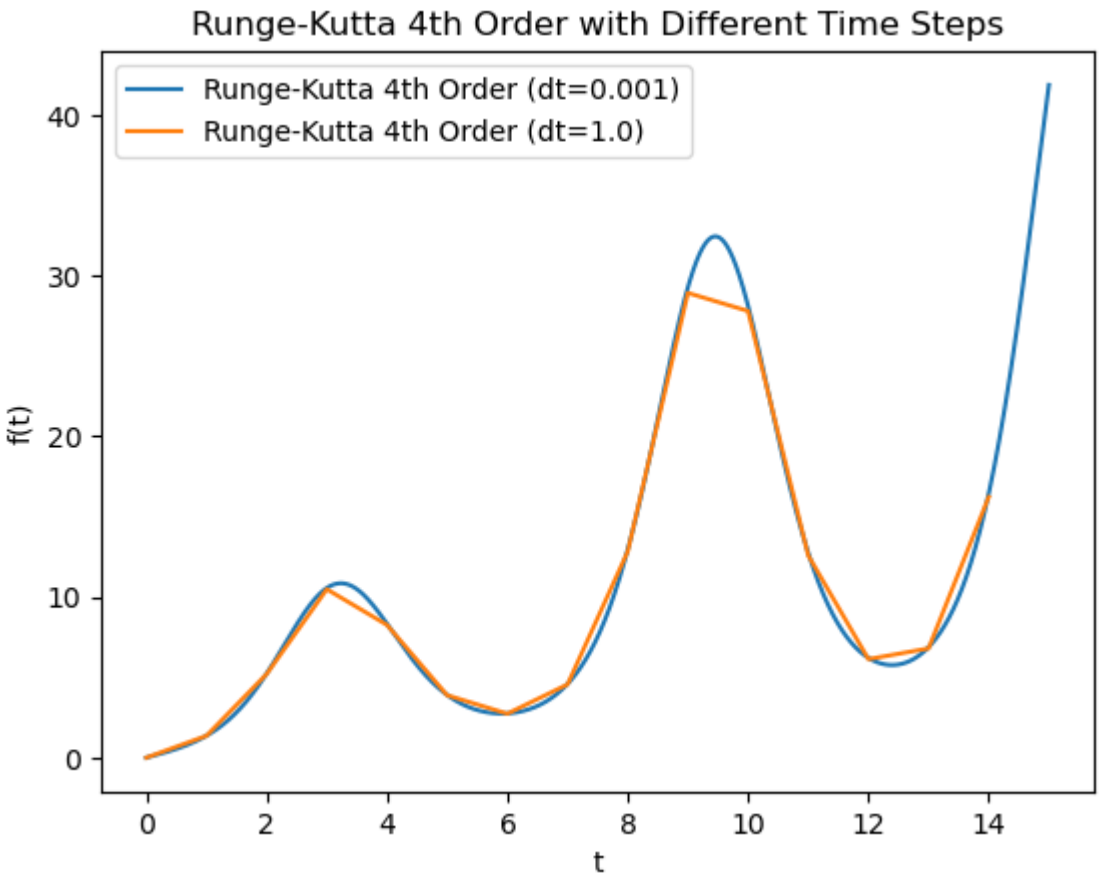


We see that the plots overlap, they give the same result

b)

Now take $\Delta t = 1.0$ and solve the equation using the Runge-Kutta method. Plot the solution on top of the result with $\Delta t = 0.001$.

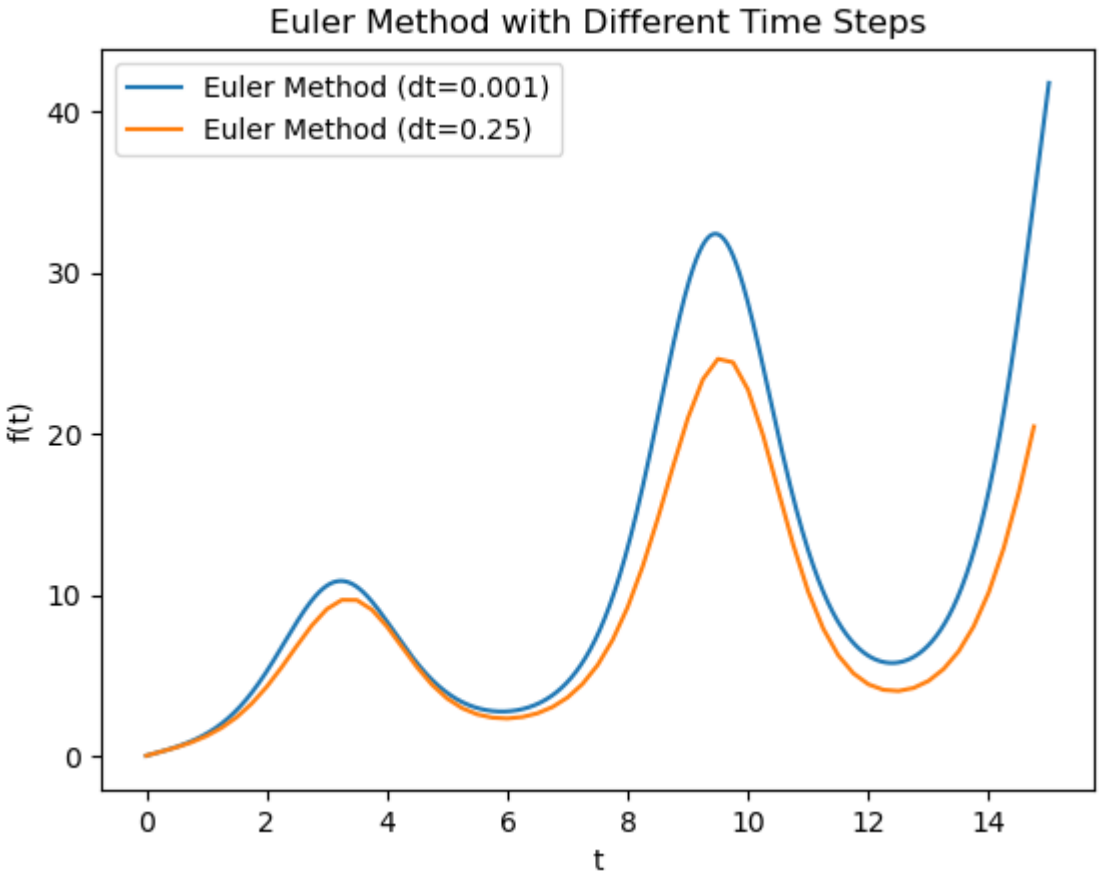
In [ ]:
```
t_rk_large, f_rk_large = runge_kutta(f_prime, 0, 1.0)

plt.plot(t_rk, f_rk, label='Runge-Kutta 4th Order (dt=0.001)')
plt.plot(t_rk_large, f_rk_large, label='Runge-Kutta 4th Order (dt=1.0)')
plt.xlabel('t')
plt.ylabel('f(t)')
plt.title('Runge-Kutta 4th Order with Different Time Steps')
plt.legend()
plt.show()
```



c)

Take $\Delta t = 0.25$ and solve the equation using the Euler method. Plot the solution on top of the result with $\Delta t = 0.001$ and comment on the result.

In [ ]:
```
t_euler_large, x_euler_large = euler_method(f_prime, 0, 0.25)

plt.plot(t_euler, x_euler, label='Euler Method (dt=0.001)')
plt.plot(t_euler_large, x_euler_large, label='Euler Method (dt=0.25)')
plt.xlabel('t')
plt.ylabel('f(t)')
plt.title('Euler Method with Different Time Steps')
plt.legend()
plt.show()
```



Like we saw 3.1 c increasing the time steps causes a decrease in accuracy when using the explicit euler method. Therefore it makes sense that using a bigger time step makes the soultion less accurate and more rigid.