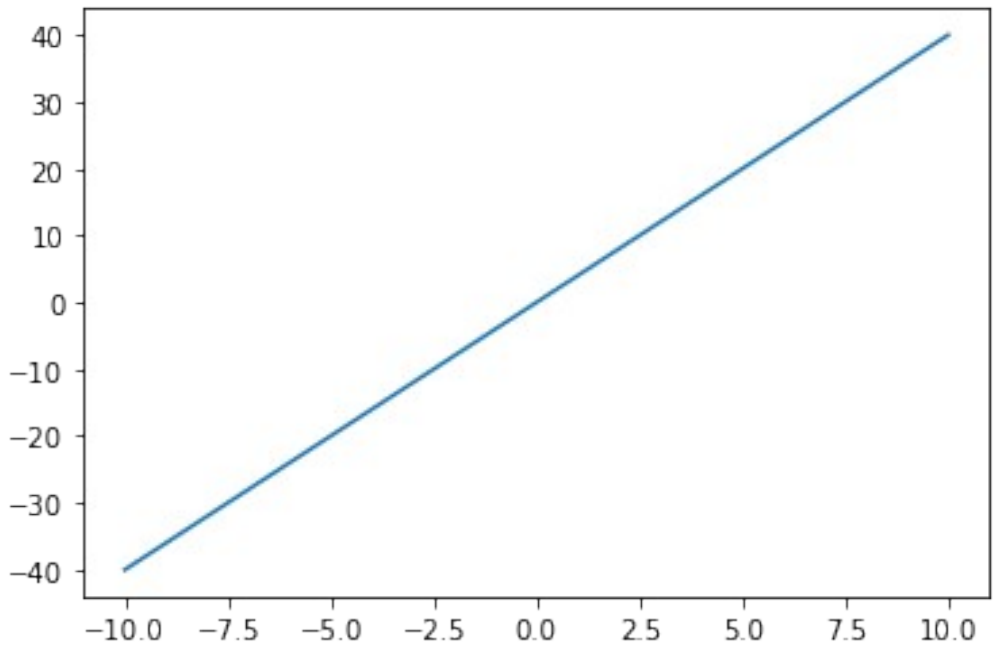
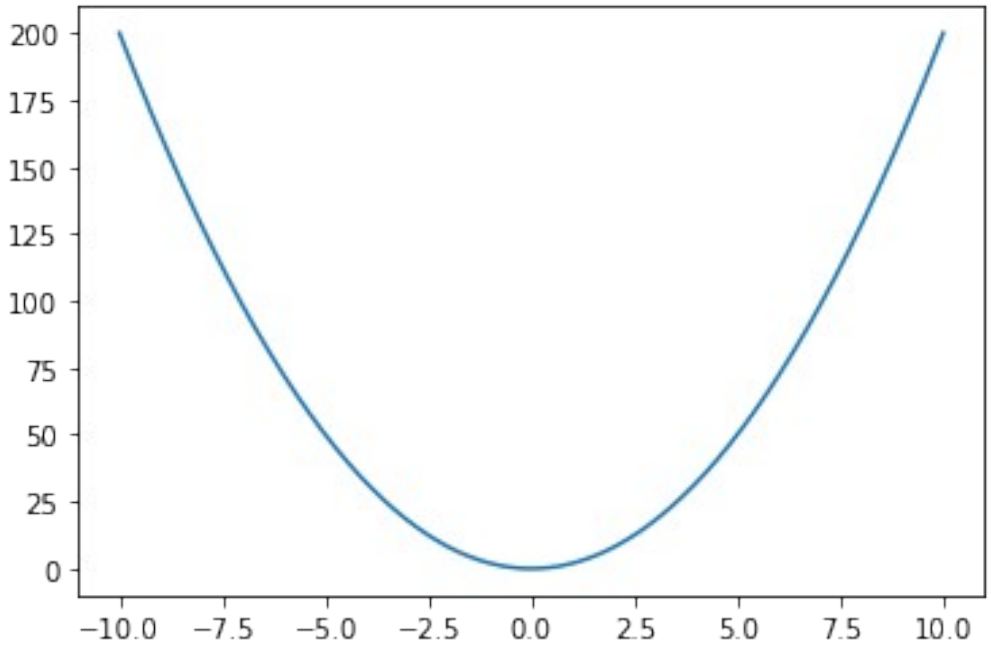
```
# Qno: 01
# Monomial plotting

#########################################################################
#####################################################################

# 1. Linear monomial function
# f(x) = 4x
# 2. Quadratic monomial function
# f(x) = 2x^2
# 3. Qubic monomial function
# f(x) = 2x^3
# 4. Quartic monomial function
# f(x) = 2x^4
# 5. Qaintic monomial function
# f(x) = 2x^5
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure()
x = np.linspace(-10, 10, 1000)
y = 4*x
plt.plot(x, y)
```

```
[<matplotlib.lines.Line2D at 0x7fc9c2e07940>]
```



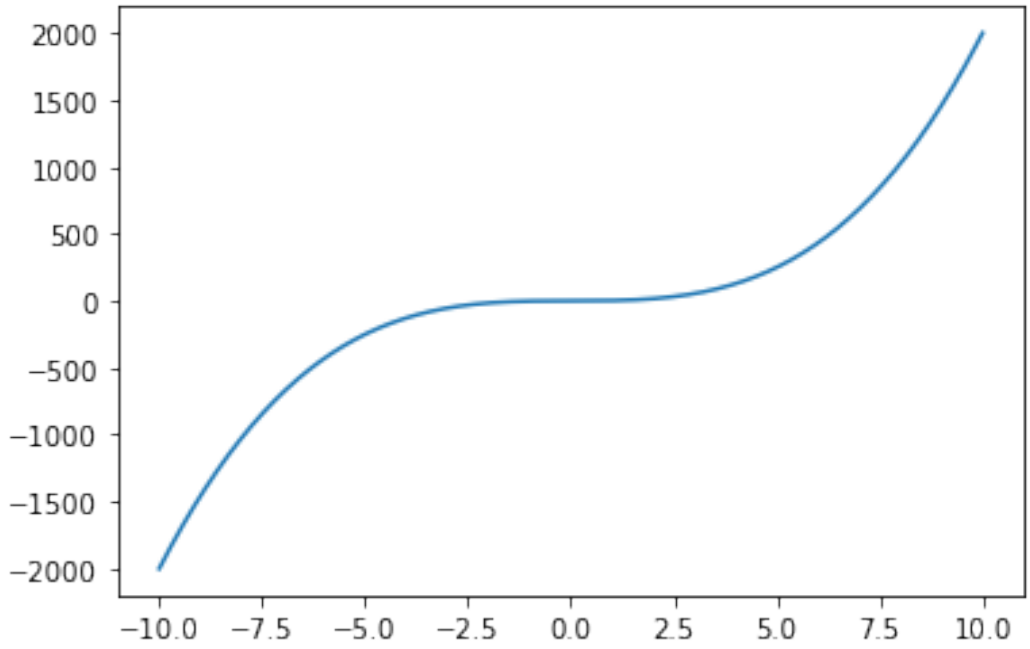
```
# Quadratic monomial function plot
x = np.linspace(-10, 10, 1000)
y = 2*x**2
plt.plot(x, y)
```

```
[<matplotlib.lines.Line2D at 0x7fc9c2dabd00>]
```



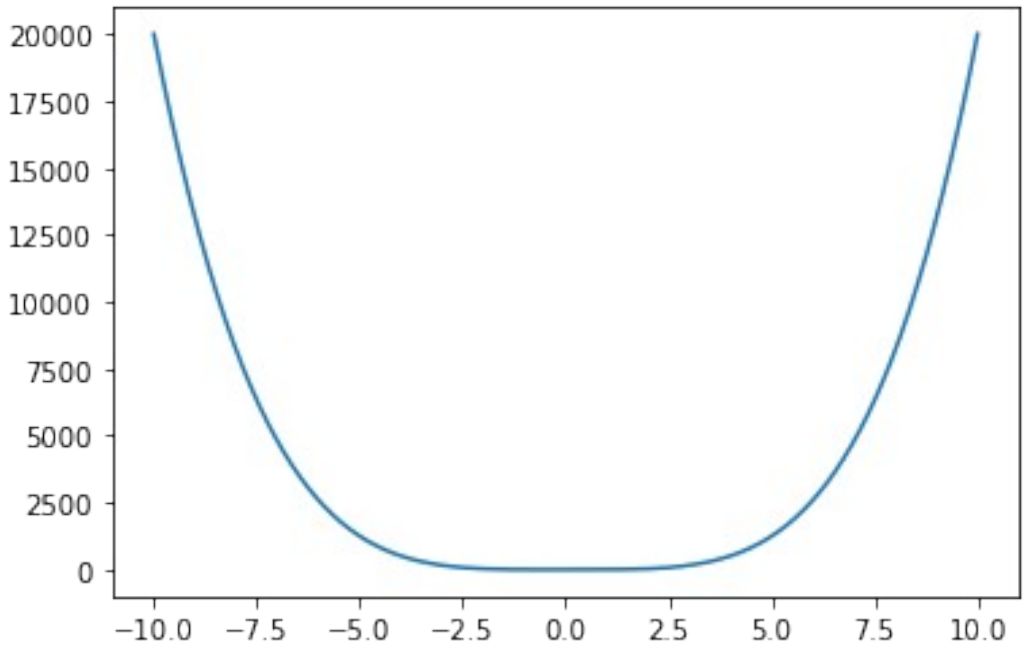

```
# Cubic monomial function plot
x = np.linspace(-10, 10, 1000)
y = 2*x**3
plt.plot(x, y)
```

```
[<matplotlib.lines.Line2D at 0x7fc9c2d168b0>]
```



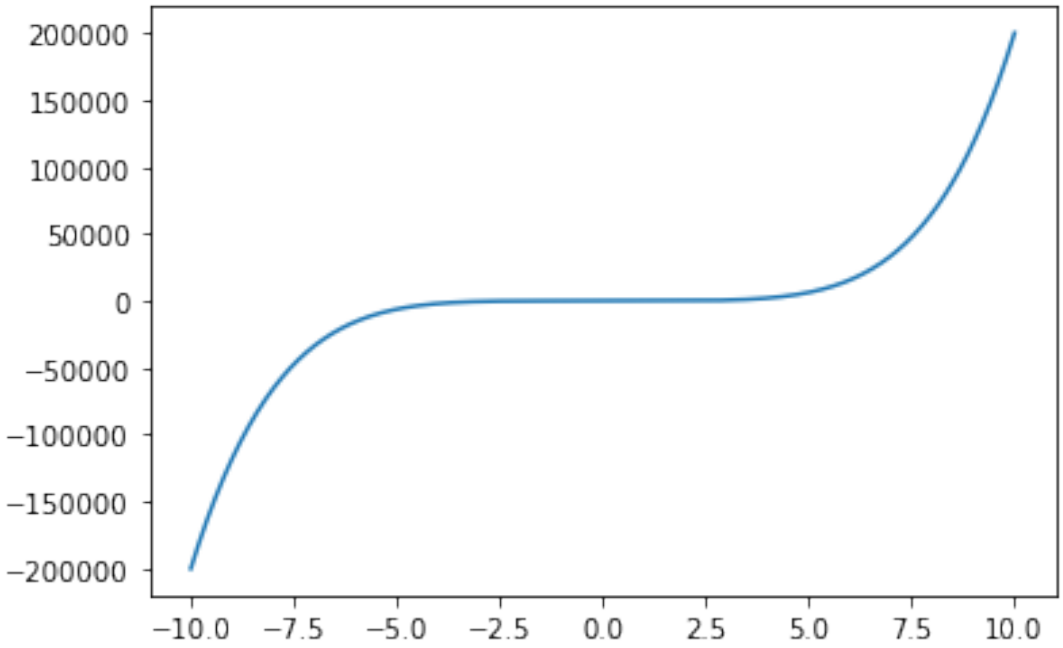

```
# Qurtic monomial function plot
x = np.linspace(-10, 10, 1000)
y = 2*x**4
plt.plot(x, y)
```

```
[<matplotlib.lines.Line2D at 0x7fc9c2cf3fa0>]
```



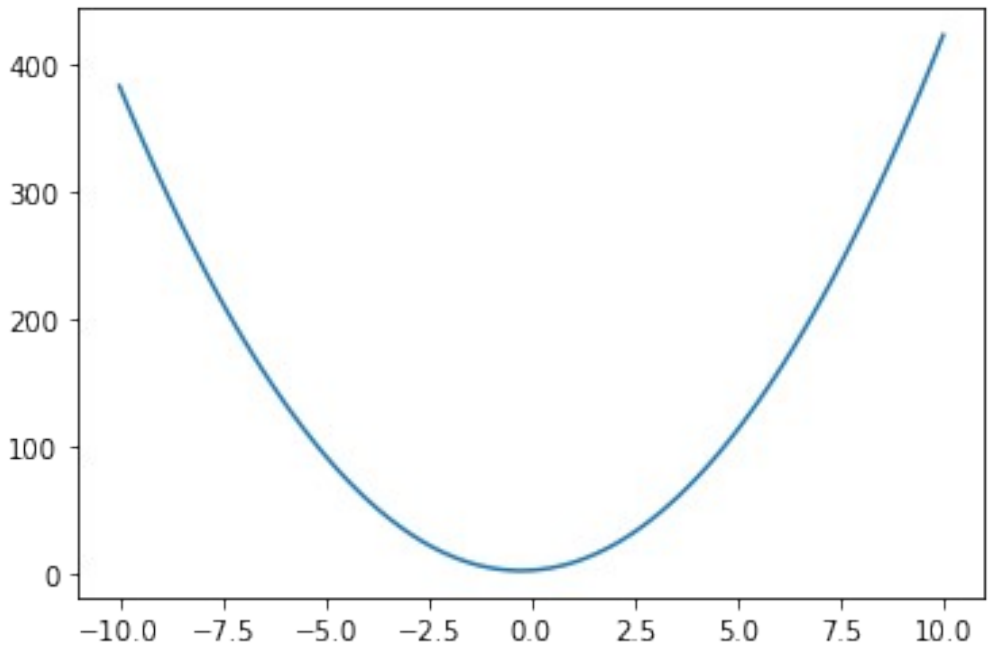
```
# Quintic monomial function plot
x = np.linspace(-10, 10, 1000)
y = 2*x**5
plt.plot(x, y)
```

```
[<matplotlib.lines.Line2D at 0x7fc9c2c5e670>]
```



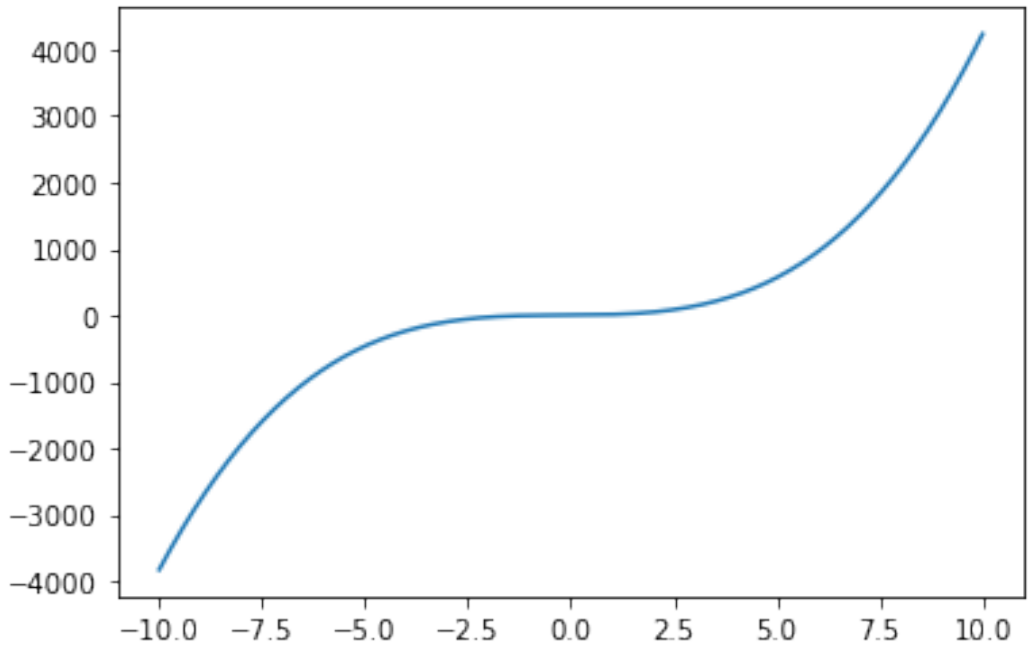

```
# Qno: 02
# Polynomial plotting

#########################################################################
#########################################################################
# 2. Quadratic Polynomial function
# f(x) = 2x^2 + 3y+3
# 3. Qubic Polynomial function
# f(x) = 2x^3+ 3y+3
# 4. Quartic Polynomial function
# f(x) = 2x^4+ 3y+3
# 5. Qaintic Polynomial function
# f(x) = 2x^5+3y+3

# Quadratic Polynomial function
x = np.linspace(-10, 10, 100)
y = 4*x**2+2*x+3
plt.plot(x, y)
```

```
[<matplotlib.lines.Line2D at 0x7fc9c2bbcfa0>]
```



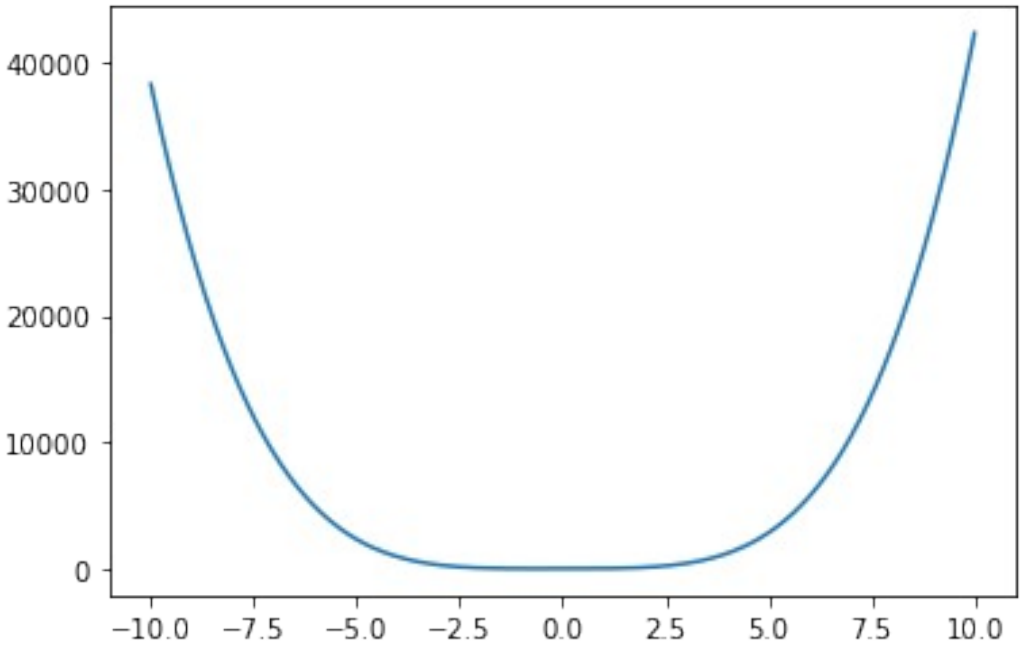

```
# Cubic Polynomial function
x = np.linspace(-10, 10, 1000)
z = 4*x**3+2*x**2+3*x+3
plt.plot(x, z)
```

```
[<matplotlib.lines.Line2D at 0x7fc9c2ba5040>]
```



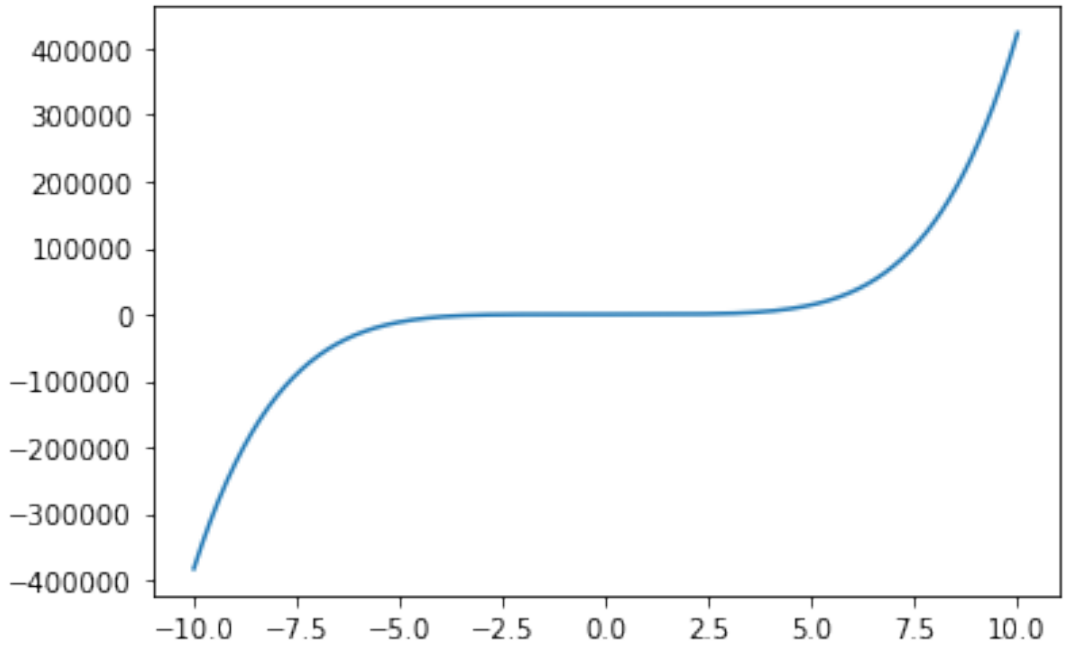
```
#Quartic Polynomial function
x = np.linspace(-10, 10, 1000)
z = 4*x**4+2*x**3+3*x**2+3*x+3
plt.plot(x, z)
```

```
[<matplotlib.lines.Line2D at 0x7fc9c2aff760>]
```



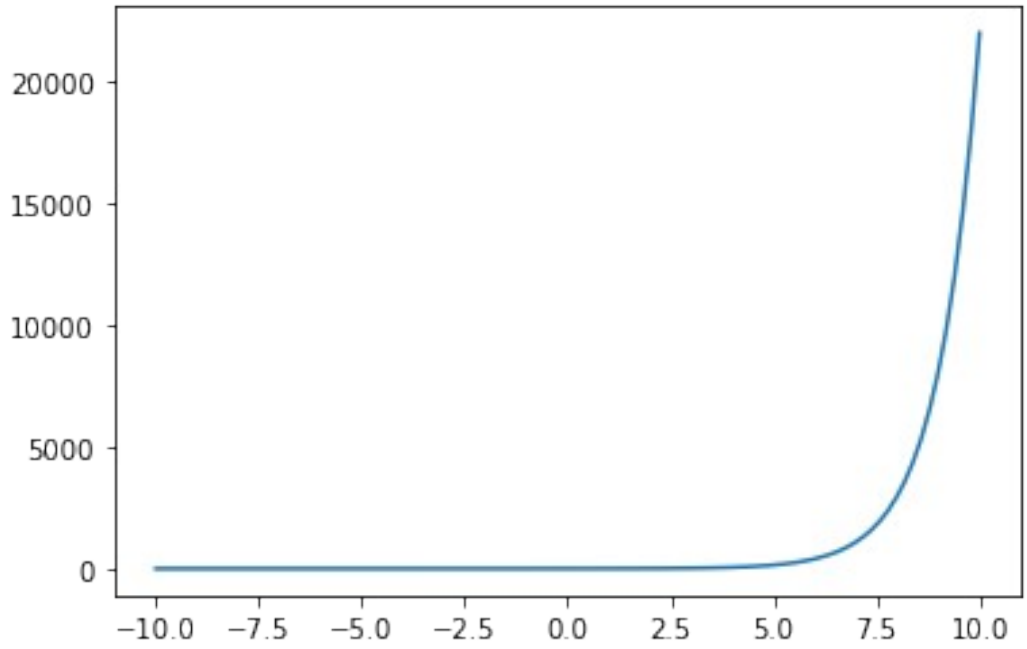

```
#Quintic Polynomial function
x = np.linspace(-10, 10, 1000)
z = 4*x**5+2*x**4+3*x**3+3*x**2+3*x+3
plt.plot(x, z)
```

```
[<matplotlib.lines.Line2D at 0x7fc9c2ad3940>]
```



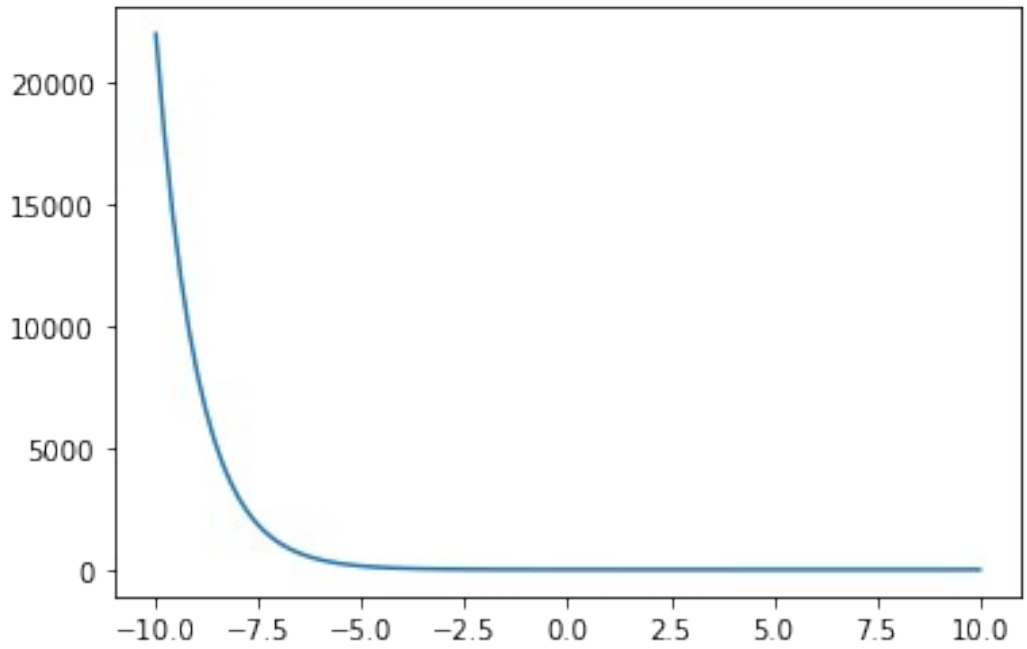

```
# Qno: 03
# Exponential Plotting
# 1. Exp(x)
# 2. Exp(-x)
# 3. log(x)
# 4. Exp(jx)
# 5. sin(x)
# 6. sin(2x)
# 7. sin(1/2x)
# 8. cos(x)
# 9. cos(2x)
# 10. cos(1/2x)

# Exp(x)
x = np.linspace(-10, 10, 100)
exp_of_x = np.exp(x)
plt.plot(x, exp_of_x)
```

```
[<matplotlib.lines.Line2D at 0x7fc9c2d5afa0>]
```




```
# Exp(-x)
x = np.linspace(-10, 10, 100)
exp_of_x = np.exp(-x)
plt.plot(x, exp_of_x)
```

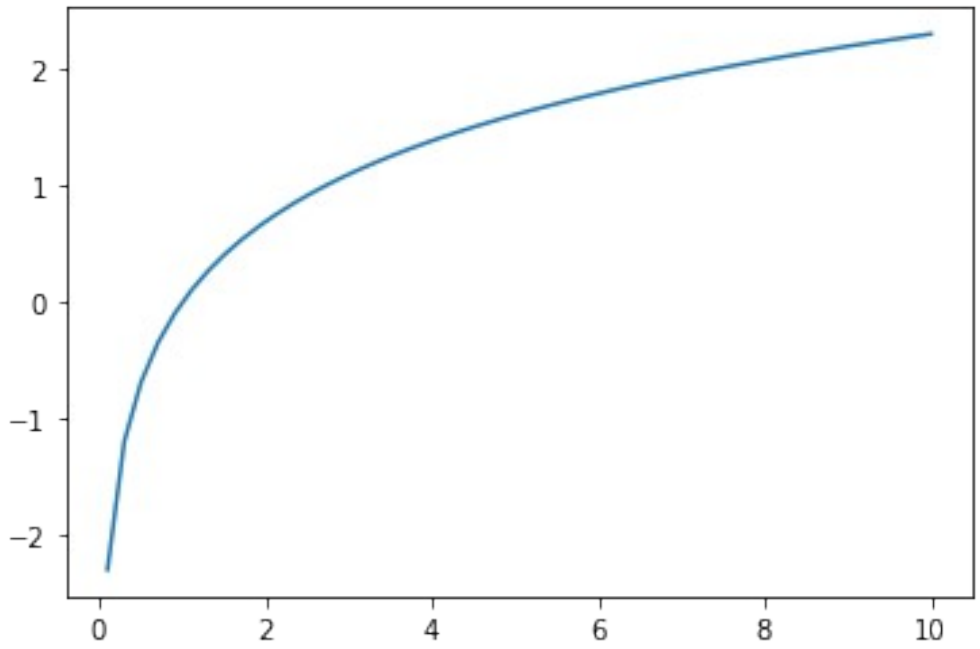
```
[<matplotlib.lines.Line2D at 0x7fc9c2b2abb0>]
```



```
# log(x)
x = np.linspace(-10, 10, 100)
exp_of_x = np.log(x)
plt.plot(x, exp_of_x)
```

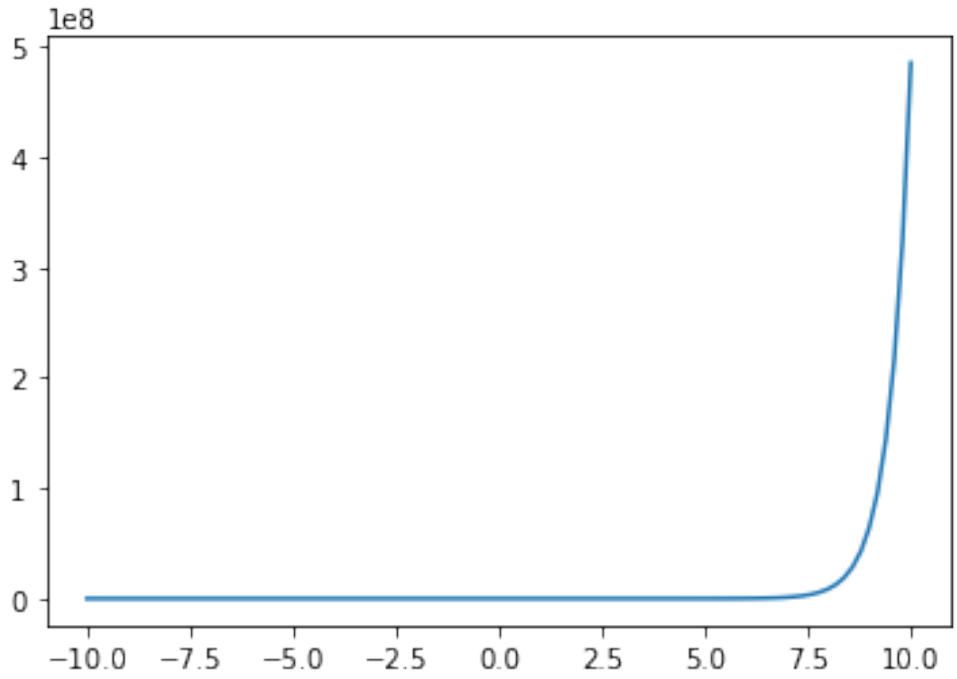
```
<ipython-input-16-3130aa0d4d49>:3: RuntimeWarning: invalid value
encountered in log
  exp_of_x = np.log(x)
```

```
[<matplotlib.lines.Line2D at 0x7fc9c29ca880>]
```



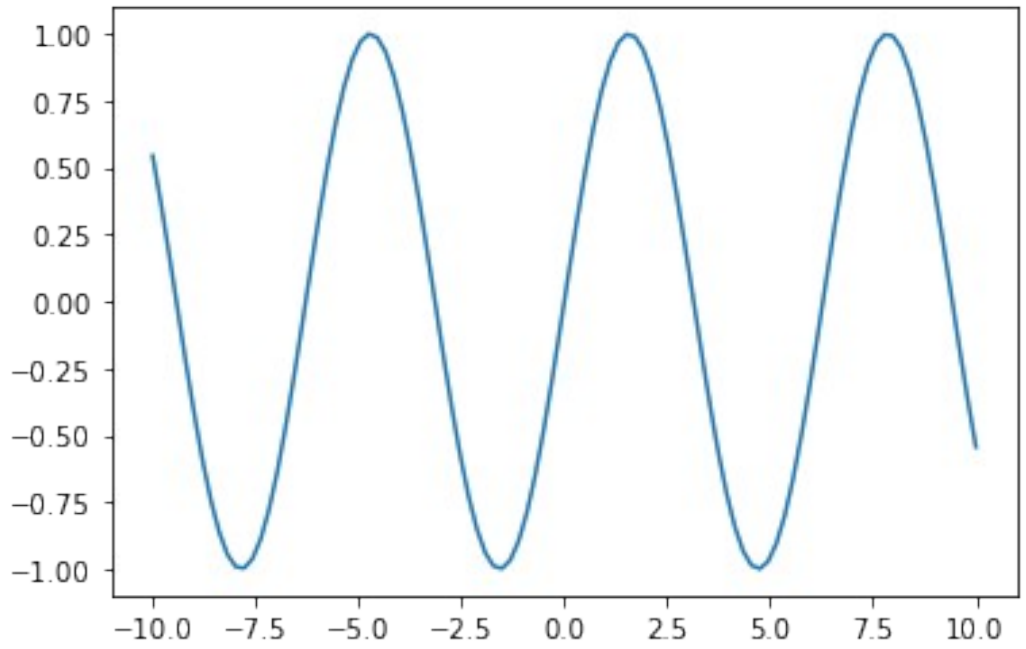

```
# Exp(jx)
x = np.linspace(-10, 10, 100)
exp_of_x = np.exp(2*x)
plt.plot(x, exp_of_x)
```

```
[<matplotlib.lines.Line2D at 0x7fc9c299ab50>]
```



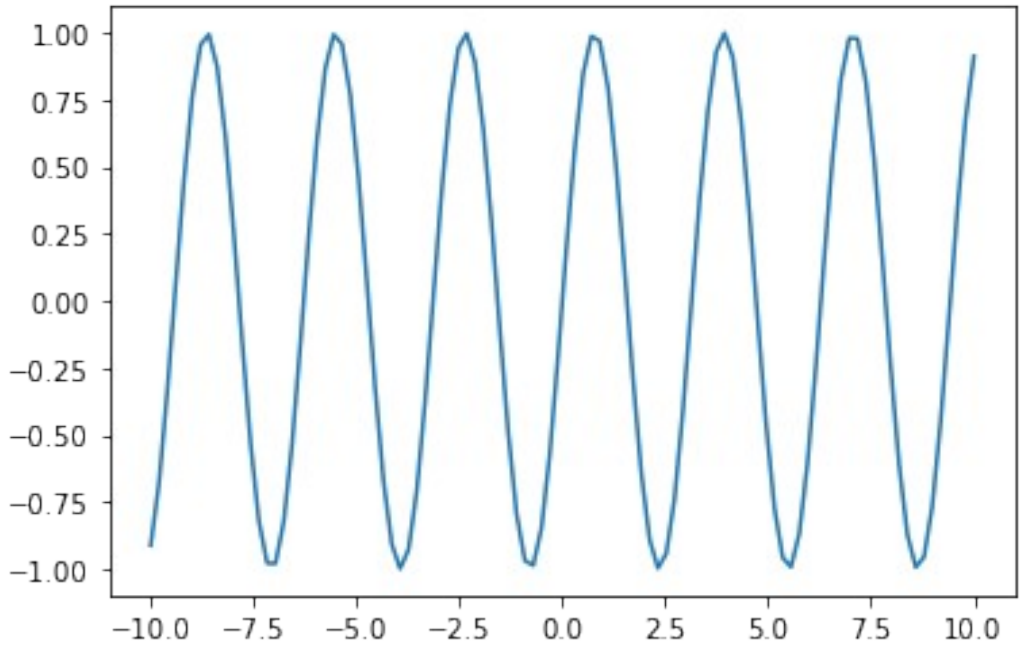

```
# sin(x)
x = np.linspace(-10, 10, 100)
sin_of_x = np.sin(x)
plt.plot(x, sin_of_x)
```

```
[<matplotlib.lines.Line2D at 0x7fc9c28ff250>]
```



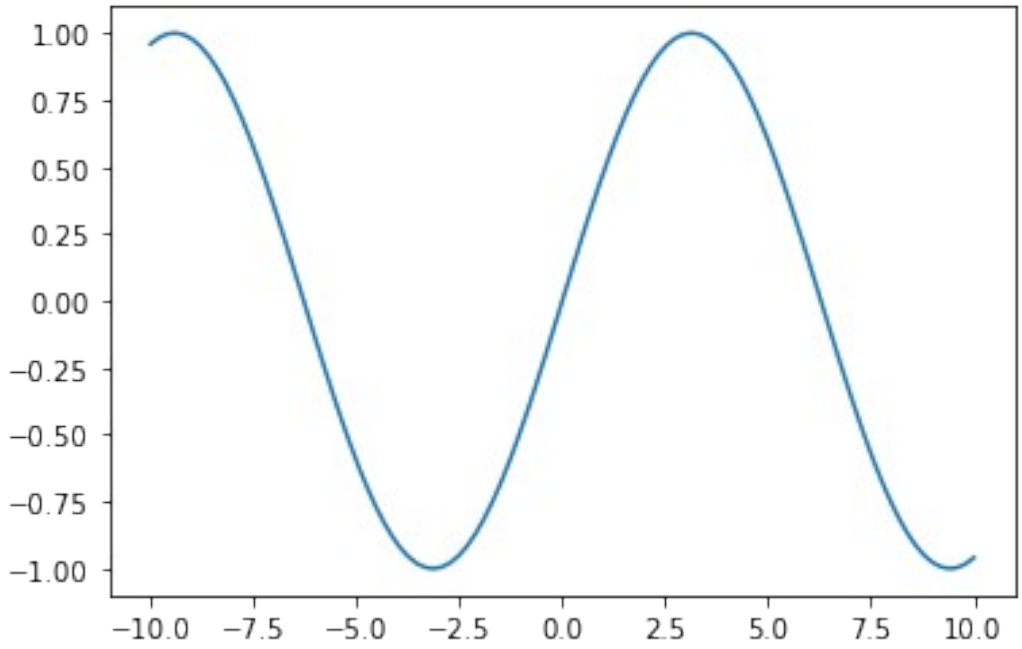
```
# sin(2x)
x = np.linspace(-10, 10, 100)
sin_of_2x = np.sin(2*x)
plt.plot(x, sin_of_2x)
```

```
[<matplotlib.lines.Line2D at 0x7fc9c28dc6a0>]
```



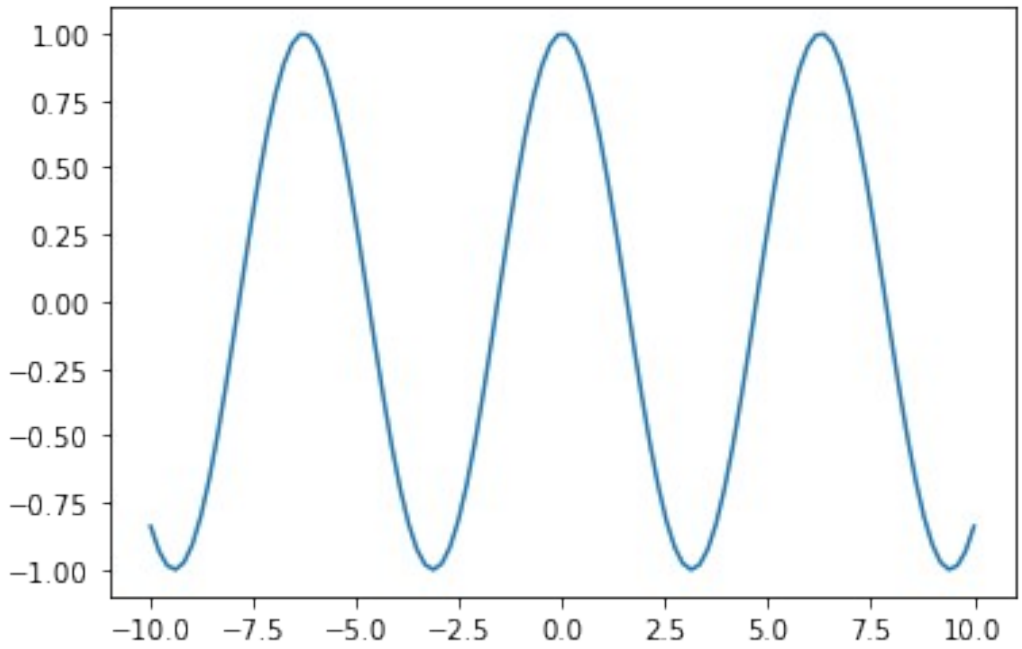

```
# sin(1/2x)
x = np.linspace(-10, 10, 100)
sin_of_half_x = np.sin(1/2*x)
plt.plot(x, sin_of_half_x)
```

```
[<matplotlib.lines.Line2D at 0x7fc9c28b9e20>]
```



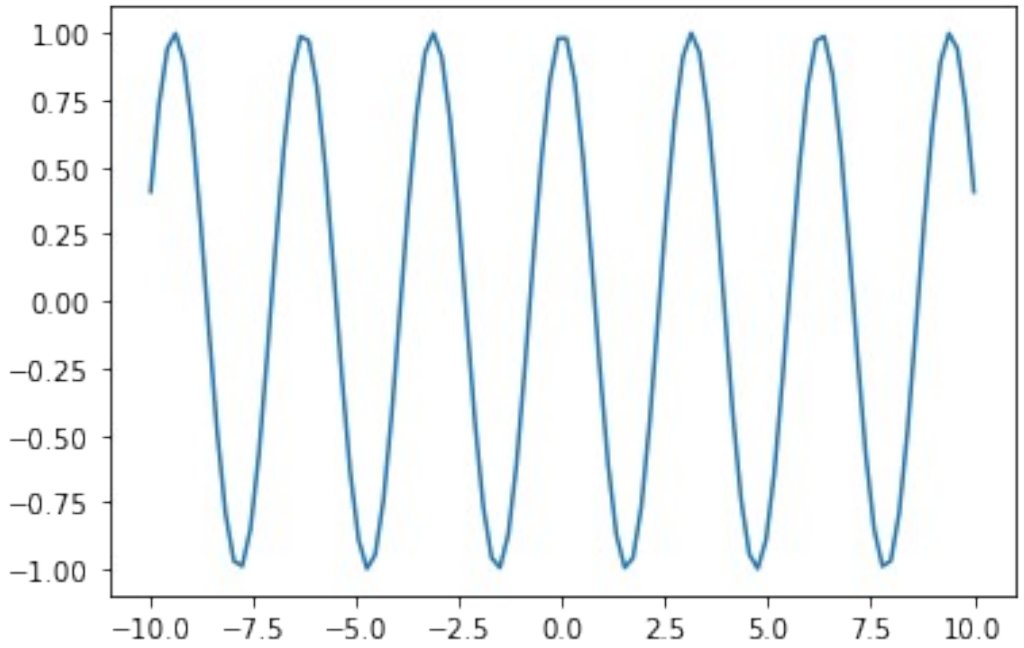

```
# cos(x)
x = np.linspace(-10, 10, 100)
cos_of_x = np.cos(x)
plt.plot(x, cos_of_x)
```

```
[<matplotlib.lines.Line2D at 0x7fc9c2822250>]
```



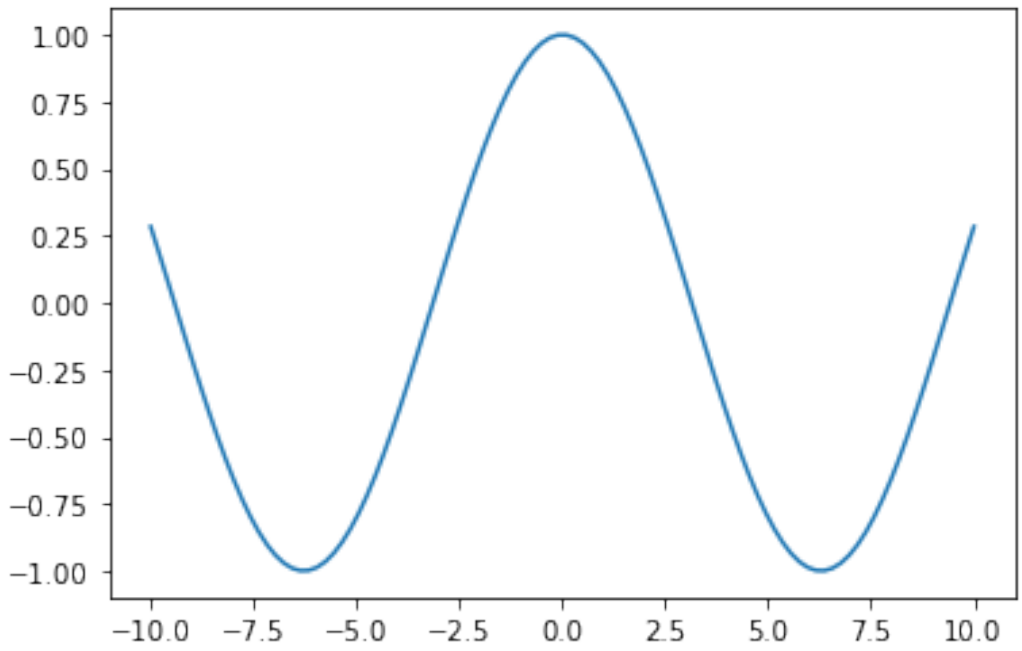
```
# cos(2x)
x = np.linspace(-10, 10, 100)
cos_of_2x = np.cos(2*x)
plt.plot(x, cos_of_2x)
```

```
[<matplotlib.lines.Line2D at 0x7fc9c277eaf0>]
```



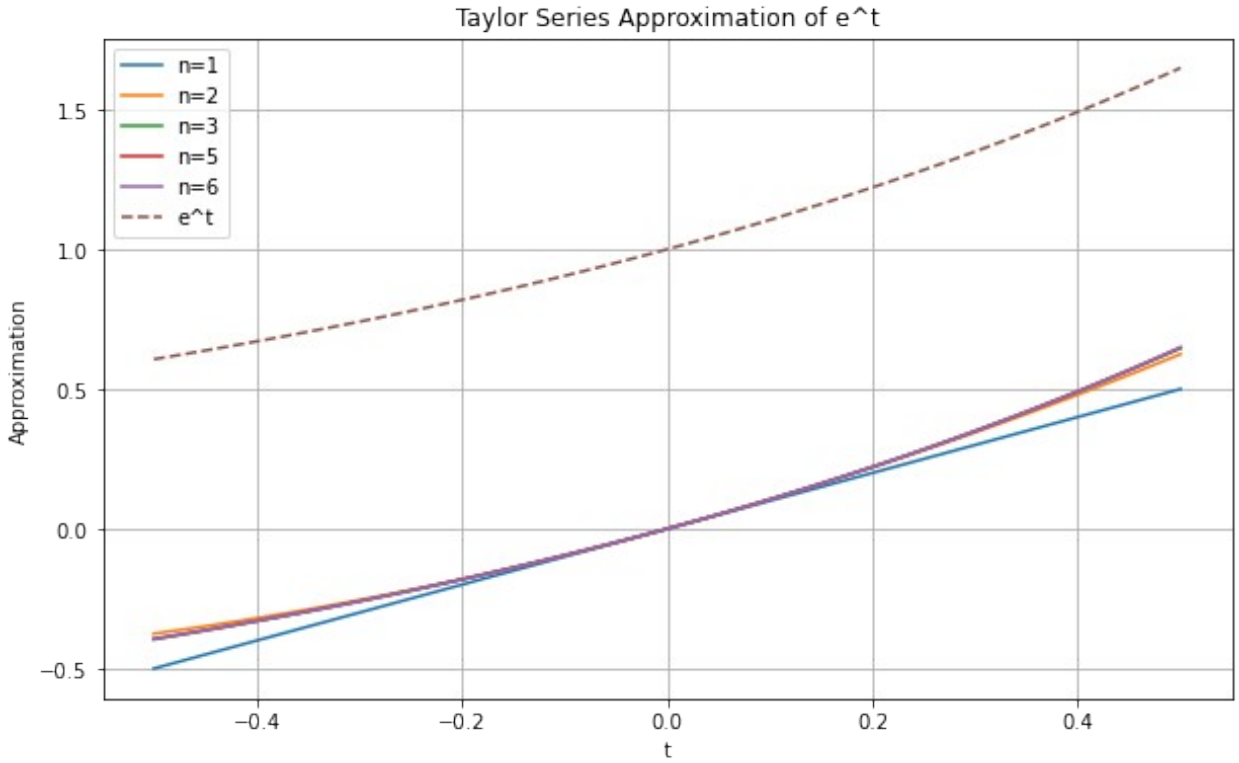

```
# cos(1/2x)
x = np.linspace(-10, 10, 100)
cos_of_half_x = np.cos(1/2*x)
plt.plot(x, cos_of_half_x)
```

```
[<matplotlib.lines.Line2D at 0x7fc9c2769160>]
```



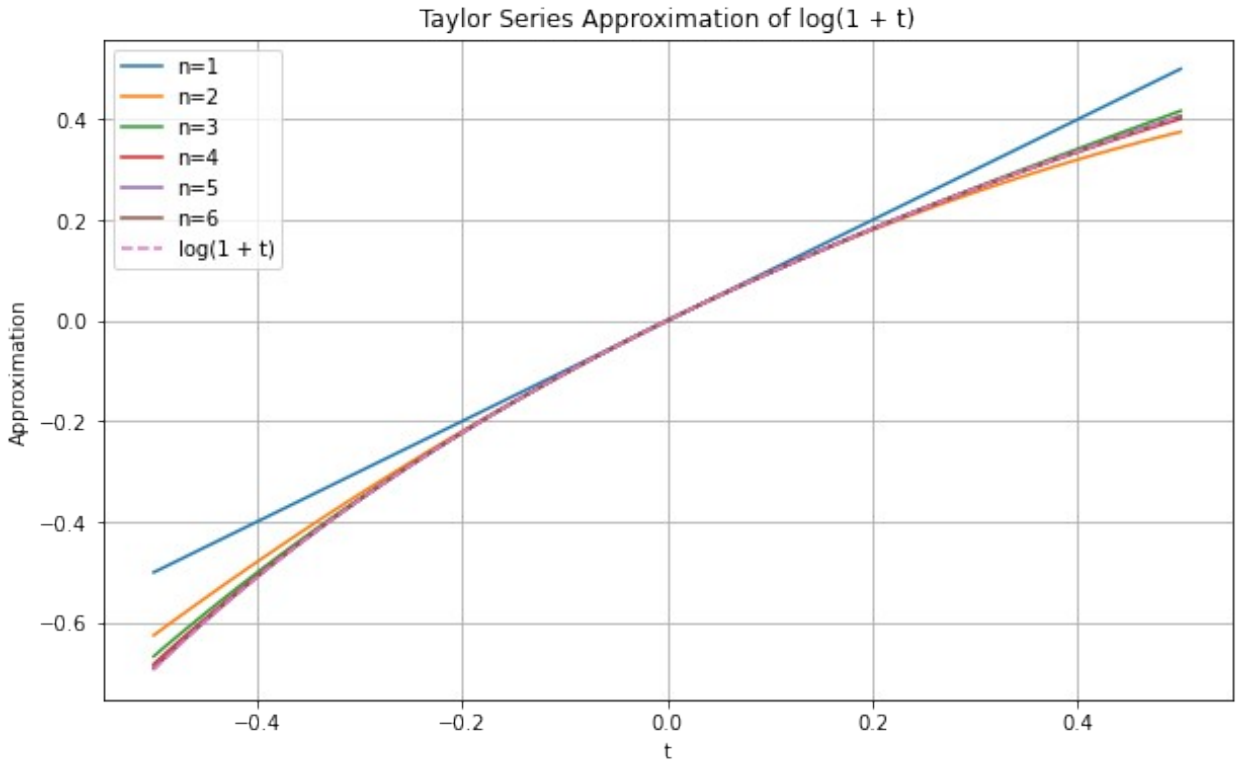

```python
#Qno:04
#(a)
def taylor_approximation(t, n):
    approximation = 0
    for k in range(1, n+1):
        approximation += (1 / np.math.factorial(k)) * (t**k)
    return approximation

# Values for t
t_values = np.linspace(-0.5, 0.5, 100)  # Adjust the range as needed

# Calculate the approximations for different values of n
n_values = [1, 2, 3, 5, 6]  # You can add more values to increase
accuracy

plt.figure(figsize=(10, 6))

for n in n_values:
    approximations = [taylor_approximation(t, n) for t in t_values]
    plt.plot(t_values, approximations, label=f'n={n}')

# Plot the actual e^t
actual_values = np.exp(t_values)
plt.plot(t_values, actual_values, label='e^t', linestyle='--')

plt.title('Taylor Series Approximation of e^t')
plt.xlabel('t')
plt.ylabel('Approximation')
```

```
plt.legend()
plt.grid(True)
plt.show()
```



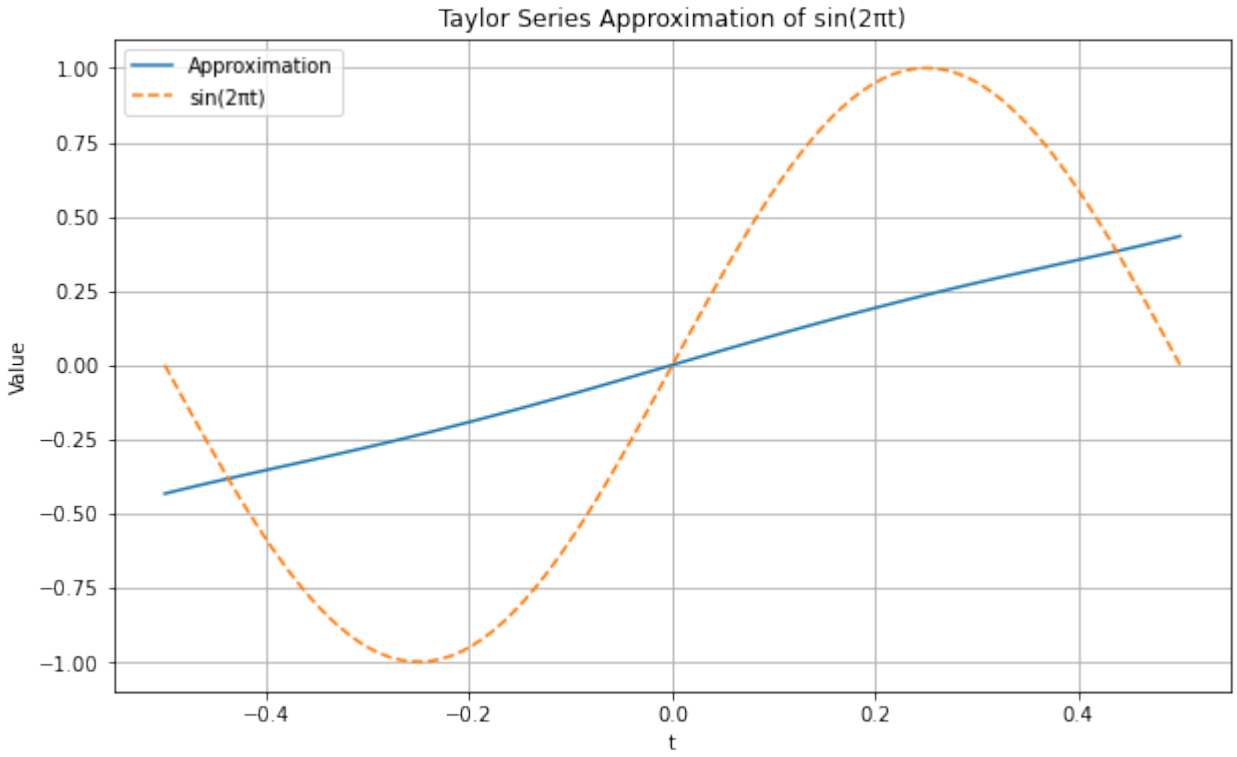

```
#Qno:04
#(b)
def taylor_approximation(t, n):
    approximation = 0
    for k in range(1, n+1):
        term = ((-1) ** (k+1)) * (t ** k) / k
        approximation += term
    return approximation

# Values for t
t_values = np.linspace(-0.5, 0.5, 100)  # Adjust the range as needed


n_values = [1, 2, 3, 4, 5, 6]  # You can add more values to increase
accuracy

plt.figure(figsize=(10, 6))

for n in n_values:
    approximations = [taylor_approximation(t, n) for t in t_values]
    plt.plot(t_values, approximations, label=f'n={n}')
```

```
# Plot the actual log(1 + t)
actual_values = np.log(1 + t_values)
plt.plot(t_values, actual_values, label='log(1 + t)', linestyle='--')

plt.title('Taylor Series Approximation of log(1 + t)')
plt.xlabel('t')
plt.ylabel('Approximation')
plt.legend()
plt.grid(True)
plt.show()
```



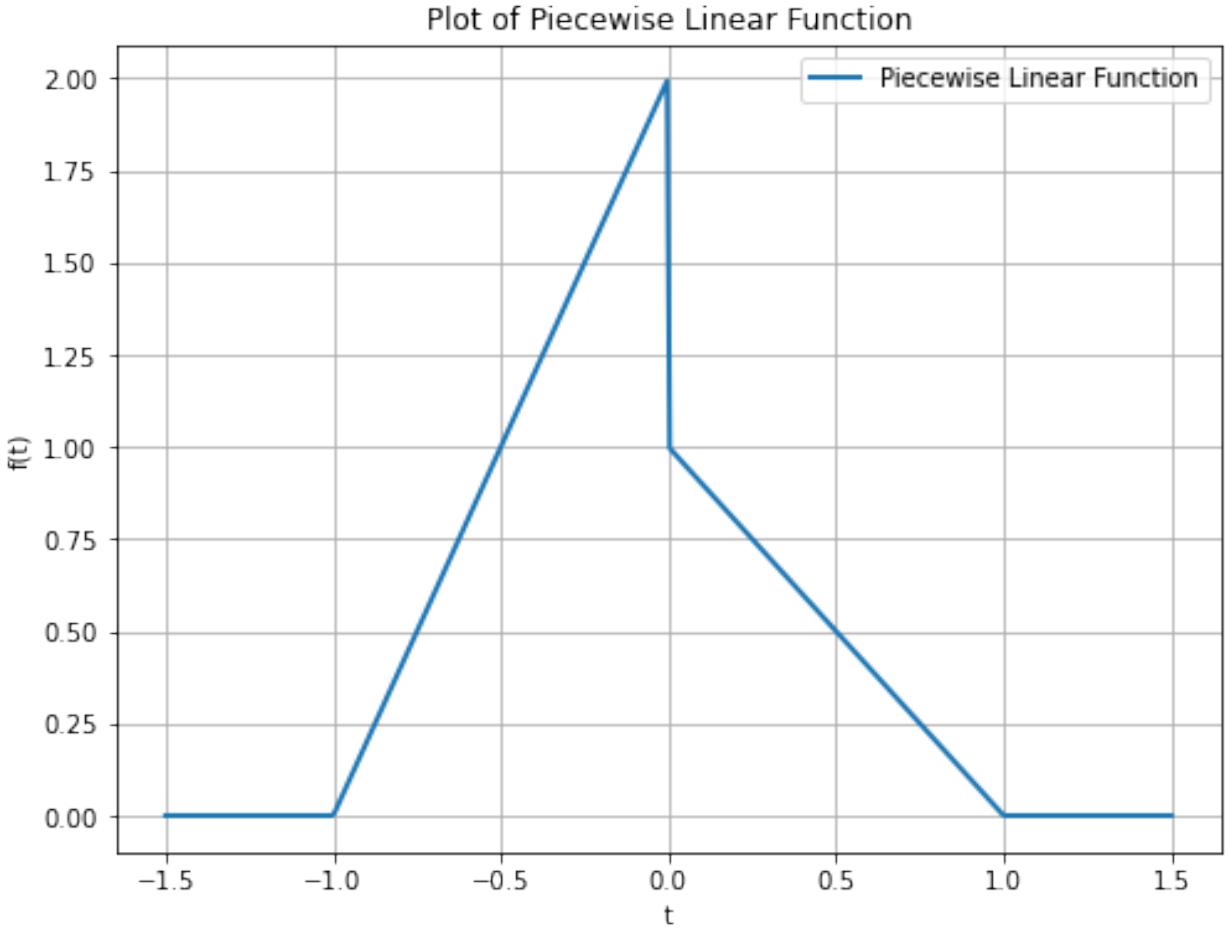

```
#Qno:04
#(c)
def taylor_approximation(t):
    approximation = t - (2 * np.pi / 6) * t**3 + (8 * np.pi**3 / 120)
* t**5
    return approximation

# Values for t
t_values = np.linspace(-0.5, 0.5, 100)  # Adjust the range as needed

# Calculate the approximations
approximations = [taylor_approximation(t) for t in t_values]

# Plot the actual sin(2πt)
```

```
actual_values = np.sin(2 * np.pi * t_values)

plt.figure(figsize=(10, 6))

plt.plot(t_values, approximations, label='Approximation')
plt.plot(t_values, actual_values, label='sin(2πt)', linestyle='--')

plt.title('Taylor Series Approximation of sin(2πt)')
plt.xlabel('t')
plt.ylabel('Value')
plt.legend()
plt.grid(True)
plt.show()
```



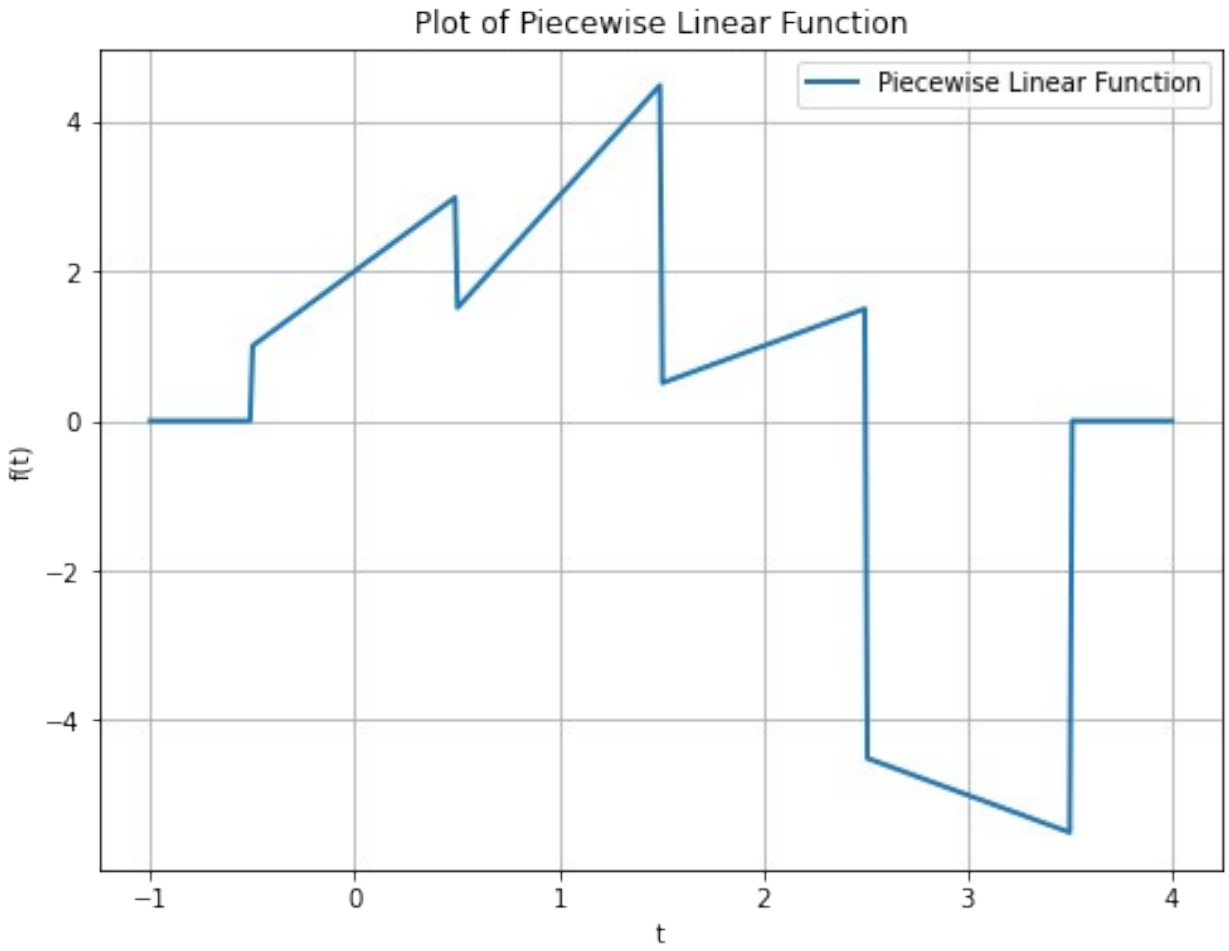

```
#Qno: 05
#piecewise plotting
#(b)
def piecewise_linear(t):
    if -1 <= t < 0:
        return 2 * (t + 1)
    elif 0 <= t <= 1:
        return 1 - t
    else:
        return 0

# Create a range of t values for plotting
```

```
t_values = np.linspace(-1.5, 1.5, 400)

# Calculate the corresponding y values for the piecewise linear
function
y_values = [piecewise_linear(t) for t in t_values]

# Plot the piecewise linear function
plt.figure(figsize=(8, 6))
plt.plot(t_values, y_values, label='Piecewise Linear Function',
linewidth=2)
plt.xlabel('t')
plt.ylabel('f(t)')
plt.legend()
plt.grid(True)
plt.title('Plot of Piecewise Linear Function')
plt.show()
```




```
#Qno:05
#(a)
def piecewise_linear(t):
```

```
    if -0.5 <= t < 0.5:
        return 2 * (t + 1)
    elif 0.5 <= t < 1.5:
        return 3 * t
    elif 1.5 <= t < 2.5:
        return t - 1
    elif 2.5 <= t < 3.5:
        return -t - 2
    else:
        return 0

# Create a range of t values for plotting
t_values = np.linspace(-1, 4, 400)

# Calculate the corresponding y values for the piecewise linear
function
y_values = [piecewise_linear(t) for t in t_values]

# Plot the piecewise linear function
plt.figure(figsize=(8, 6))
plt.plot(t_values, y_values, label='Piecewise Linear Function',
linewidth=2)
plt.xlabel('t')
plt.ylabel('f(t)')
plt.legend()
plt.grid(True)
plt.title('Plot of Piecewise Linear Function')
plt.show()
```




```python
#Qno:05
#(c)
def b2(t):
    if -3/2 <= t <= -1/2:
        return (t + 3/2)**2 / 2
    elif -1/2 <= t <= 1/2:
        return -t**2 + 3/4
    elif 1/2 <= t <= 3/2:
        return (t - 3/2)**2 / 2
    else:
        return 0

# Define the values of alpha_k for a finite number of terms
alphas = [1, -1, 1, -1, 1]

# Define the piecewise function f(t) using a finite number of terms
def f(t, num_terms):
    result = 0
    for k in range(-num_terms, num_terms + 1):
        result += alphas[k] * b2(t - k)
```

```
    return result

# Create a range of t values for plotting
t_values = np.linspace(-3, 3, 400)

# Specify the number of terms to include in the approximation
num_terms = 2  # You can increase this value for more terms

# Calculate the corresponding y values for the approximation
y_values = [f(t, num_terms) for t in t_values]

# Plot the piecewise function approximation
plt.figure(figsize=(8, 6))
plt.plot(t_values, y_values, label=f'Approximation ({2*num_terms+1}
terms)', linewidth=2)
plt.xlabel('t')
plt.ylabel('f(t)')
plt.legend()
plt.grid(True)
plt.title('Plot of Piecewise Function Approximation')
plt.show()
```