

PHYS 181/281, 18 Sept 2024, Dr. Kuntal Talit

Numerical Differentiation

Estimated time for the class: **75** minutes

Objectives

After completing this class, you will be able to:

1. Approximate derivatives using forward, backward, and central difference methods.
2. Differentiate between truncation and roundoff errors.
3. Analyze the trade-off between truncation and roundoff errors to find an optimal step size h.
4. Create and interpret error plots for numerical differentiation.
5. Apply numerical differentiation concepts to practical real-world problems.

Setup

In [1]:
```
## Import all the libraries, if not present, install them.
import numpy as np
import pandas as pd
from numpy import cos, sin,pi
import plotly.graph_objects as go
from math import factorial
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation, PillowWriter
from IPython.display import display, HTML
```

Pre-Assessment

- Please take the pre-assessment questions and try to answer at best of your knowledge
- choose directly **I don't know** if you dont understand the question.
- Click in the below **link** to start the assessment.
- https://forms.office.com/Pages/ResponsePage.aspx?id=o-nf9HjKmEm3p-bQaS6s5vHw702fgfxLigWNapFeBOxUN0RGTEk0MFc0MFQ4Sk5IWURQWEZXSTBWNC4u

Check student's response online

- https://forms.office.com/Pages/DesignPageV2.aspx?origin=NeoPortalPage&subpage=design&id=o-nf9HjKmEm3p-bQaS6s5vHw702fgfxLigWNapFeBOxUN0RGTEk0MFc0MFQ4Sk5IWURQWEZXSTBWNC4u&analysis=false&top

Engage (10 minutes)

- In this section you will be able to :
  - See different examples where numerical differentiation plays a key role.
  - understand the importance of numerical differentiantiation.
  - understand the difficulty of finding maxima and minima of any data.

Examples

 No description has been provided for this image
Trend in stock market data

 No description has been provided for this image
Potential energy landscape

 No description has been provided for this image
An arbitrary function

2. Explore (15 minutes)

Can you come up with some interesting ideas where numerical differentiation would be important?

- https://forms.office.com/Pages/ResponsePage.aspx?id=o-nf9HjKmEm3p-bQaS6s5vHw702fgfxLigWNapFeBOxUMk4wMFFPSDJZOElFQTVDMEhQRlRVVzZJMi4u

In [ ]:

Theory:

Taylor Expansion is a mathematical technique used to approximate a function as an infinite sum of its derivatives at a specific point. It expresses a function $f(x)$ as a polynomial based on the function's values and its derivatives at a particular point a.

The general form of the Taylor series expansion of $f(x)$ around $x = a$ is:

$$f(x) = f(a) + f'(a)(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f^{(3)}(a)}{3!}(x-a)^3 + \cdots$$

Here, $f'(a)$, $f''(a)$, etc., are the derivatives of $f(x)$ at $x = a$.

This expansion allows us to approximate complex functions using simpler polynomials, particularly near the point a.

- We have a function $f(x) = \sin(x)$. Run the below code to see what happen when we use Taylor expansion at $x = 0$.
- How different orders are important here?

In [2]:
```
# import numpy as np
# import plotly.graph_objects as go
# from math import factorial
# from numpy import cos, sin,pi
# import matplotlib.pyplot as plt


# # Define the function you want to expand (sin(x))
# def f(x):
#     return np.sin(x)

# # Define the nth derivative of sin(x)
# def nth_derivative_f(x, n):
#     if n % 4 == 0:  # 0th, 4th, 8th derivative of sin(x) is sin(x)
#         return np.sin(x)
#     elif n % 4 == 1:  # 1st, 5th, 9th derivative of sin(x) is cos(x)
#         return np.cos(x)
#     elif n % 4 == 2:  # 2nd, 6th, 10th derivative of sin(x) is -sin(x)
#         return -np.sin(x)
#     elif n % 4 == 3:  # 3rd, 7th, 11th derivative of sin(x) is -cos(x)
#         return -np.cos(x)

# # Calculate Taylor series expansion of sin(x) up to a given order
# def taylor_series_expansion(x, a, order):
```

```
#     taylor_sum = 0
#     for n in range(order + 1):  # From 0th to the specified order
#         nth_term = (nth_derivative_f(a, n) * (x - a)**n) / factorial(n)
#         taylor_sum += nth_term
#     return taylor_sum

# # Set up x values for plotting and the point of expansion (a)
# x_values = np.linspace(-2*np.pi, 2*np.pi, 10000)  # Points to evaluate the series
# a = 0  # Point of expansion (for sin(x), we use Maclaurin series, so a = 0)

# # Create a figure
# fig = go.Figure()

# # Add the actual function sin(x) (exact values)
# actual_values = f(x_values)
# fig.add_trace(go.Scatter(x=x_values, y=actual_values, mode='lines', name='sin(x) (Exact)', li

# # Create traces for each Taylor series order up to the 8th order
# for order in range(0, 12):
#     taylor_approx_values = taylor_series_expansion(x_values, a, order)
#     visible = [False] * 12  # Set all initially as not visible
#     visible[0] = True      # Always keep the actual function visible
#     visible[order] = True  # Set the corresponding Taylor series visible
#     fig.add_trace(go.Scatter(x=x_values, y=taylor_approx_values, mode='lines', name=f'Taylor

# # Create steps for the slider (from 1st to 8th order)
# steps = []
# for order in range(1, 12):
#     step = dict(
#         method="update",
#         args=[{"visible": [True if i == 0 or i == order else False for i in range(13)]},  # C
#               {"title": f"Taylor Series Expansion of sin(x) (Order {order})"}],
#         label=f"{order}"
#     )
#     steps.append(step)

# # Add a slider to control the order of the Taylor series
# sliders = [dict(
#     active=0,
#     currentvalue={"prefix": "Order: "},
#     pad={"t": 50},
#     steps=steps
# )]

# # Update the layout with the slider
# fig.update_layout(
#     sliders=sliders,
#     title="Taylor Series Expansion of sin(x)",
#     xaxis_title="x",
#     yaxis_title="f(x)",
#     showlegend=True,
#     width=800,  # Set the width of the figure
#     height=600,   # Set the height of the figure
#     # Fix the x and y axis limits
#     xaxis=dict(range=[-np.pi, np.pi]),  # Set the x-axis limits (e.g., -2π to 2π)
#     yaxis=dict(range=[-2, 2]),              # Set the y-axis limits (e.g., -2 to 2)

# )

# # Show the interactive plot
# fig.show()


# ### Question to thinks of:
# - what happens when we increase the order?
# - why some 'orders' are improving the fitting while some are not?
# - **hint:** think of Taylor expansion of sin(x) at x=0.
```

3. Explain (20 minutes)

Objective: To understand the mathematical foundations of different numerical differentiation methods.

Theory:

Let us see how we can simply use Taylor series to find out different numerical differentiation techniques.

Board work: How I can arrive at different numerical differentiation formulas using Taylor series?



The derivative of a function at a point provides a measure of how the function's output changes as its input changes. More formally, the derivative of a function $f(x)$ at a particular point x gives the **instantaneous rate of change** of the function with respect to x.

Mathematically, the derivative is defined as the limit of the average rate of change of the function over an interval as the interval becomes infinitely small.

For a function $f(x)$, the derivative is defined as:

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

The derivative of $f(x)$ at $x = a$ is given by:

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

Where: - $f'(a)$ is the derivative of $f(x)$ at $x = a$, - h is a small change in x, - $f(a + h)$ is the function value at $x = a + h$, - $f(a)$ is the function value at $x = a$.
This equation represents the **slope of the tangent line** to the curve of $f(x)$ at the point $x = a$.

General Notation:

- The derivative of $f(x)$ with respect to x is often written as $\frac{df}{dx}$, $f'(x)$, or $D_x f(x)$.

Different numerical methods to calculate derivative

1. Forward Difference Method

The forward difference approximation uses the function value at a point x and the function value at a slightly shifted point $x + h$:

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

This method estimates the derivative at x by looking forward, considering the change in the function value over the interval h.

- **Accuracy**: First-order accurate, meaning the error is proportional to h.
- **Error Term**: The truncation error is of order $O(h)$, coming from the Taylor series expansion.

2. Backward Difference Method

The backward difference approximation looks at the difference between the function value at x and the function value at $x - h$:

$$f'(x) \approx \frac{f(x) - f(x-h)}{h}$$

This method estimates the derivative at x by looking backward, considering the change over the interval h in the opposite direction.

- **Accuracy**: First-order accurate, meaning the error is proportional to h.
- **Error Term**: The truncation error is of order $O(h)$, similar to the forward difference.

3. Central Difference Method

The central difference method uses function values at both $x + h$ and $x - h$ to approximate the derivative at x:

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

This method takes an average of the forward and backward differences, providing a more accurate estimate.

- **Accuracy**: Second-order accurate, meaning the error is proportional to h^2, which makes it more accurate than forward or backward differences for small h.
- **Error Term**: The truncation error is of order $O(h^2)$.

Second Derivative Using Central Difference

The central difference method can be used to approximate the second derivative of a function $f(x)$ by utilizing values of the function at points symmetrically distributed around x.

1. **Formula for Second Derivative:**

The central difference formula for the second derivative of a function $f(x)$ is given by:

$$f''(x) \approx \frac{f(x+h) - 2f(x) + f(x-h)}{h^2}$$

Where: - $f(x)$ is the function at point x, - h is the step size (the distance between neighboring points), - $f(x + h)$ is the function value one step forward from x, - $f(x - h)$ is the function value one step backward from x.
This formula is **second-order accurate**, meaning the error term is proportional to $O(h^2)$.

2. **Derivation:**

The formula can be derived from the **Taylor series expansion** of $f(x + h)$ and $f(x - h)$ around x.

- Taylor expansion for $f(x + h)$:
$$f(x + h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f^{(3)}(x) + O(h^4)$$
- Taylor expansion for $f(x - h)$:
$$f(x - h) = f(x) - hf'(x) + \frac{h^2}{2}f''(x) - \frac{h^3}{6}f^{(3)}(x) + O(h^4)$$
Adding these two expansions together to cancel the first-order derivative terms:
$$f(x + h) + f(x - h) = 2f(x) + h^2 f''(x) + O(h^4)$$
Rearranging to solve for $f''(x)$:
$$f''(x) \approx \frac{f(x+h) - 2f(x) + f(x-h)}{h^2}$$
Thus, the second derivative is approximated by the values of the function at $x + h$, x, and $x - h$ using the central difference formula.

Activity

- Run the below code block and explore different methods by varying h.
- Work in groups and come up with possible answers to the following questions mention below the graph.

In [3]:
```
## Define your function here
def f(t):
    return sin(t)

## define your exact derivative function here
def dfdt_E(t):
    return cos(t)
```

```
## Forward difference
def dfdt_F(f, t, dt):
    return (f(t+dt) - f(t))/dt

## Backward difference
def dfdt_B(f, t, dt):
    return (f(t) - f(t-dt))/dt

## Central difference
def dfdt_C(f, t, dt):
    return ((f(t+dt) - f(t-dt))/(2*dt))

t= np.linspace(0,2*pi,100)
dt =0.5
E= dfdt_E(t)
F= dfdt_F(f,t,dt)
B= dfdt_B(f,t,dt)
C= dfdt_C(f,t,dt)

# Create a figure
fig = go.Figure()

# Add traces for the initial plot
fig.add_trace(go.Scatter(x=t, y=f(t), mode='lines', name='f(t)', line=dict(color='black')))
fig.add_trace(go.Scatter(x=t, y=E, mode='lines', name="Exact f'(t)", line=dict(color='red')))
fig.add_trace(go.Scatter(x=t, y=F, mode='lines', name="Forward", line=dict(color='orange')))
fig.add_trace(go.Scatter(x=t, y=B, mode='lines', name="Backward", line=dict(color='green')))
fig.add_trace(go.Scatter(x=t, y=C, mode='lines', name="Central", line=dict(color='blue')))

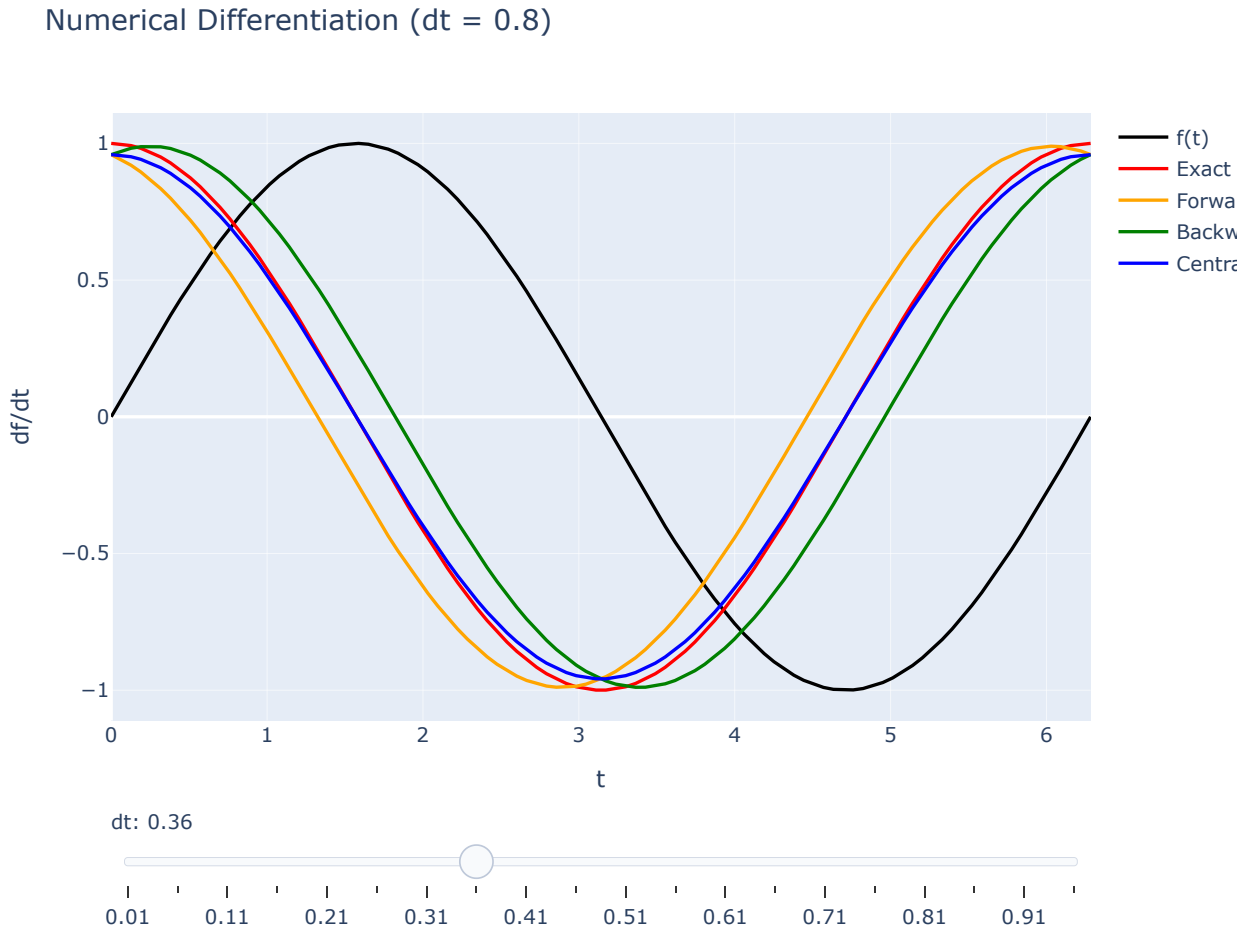
# Create a slider for varying dt
steps = []
for dt in np.arange(0.01, 1, 0.05):
    F_new = dfdt_F(f, t, dt)
    B_new = dfdt_B(f, t, dt)
    C_new = dfdt_C(f, t, dt)

    step = dict(
        method="update",
        args=[{"y": [f(t), E, F_new, B_new, C_new]},  # Update y-values of all traces
              {"title": f"Numerical Differentiation (dt = {dt:.2f})"}],
        label=f"{dt:.2f}"
    )
    steps.append(step)

# Add slider to layout
sliders = [dict(
    active=7,  # Start with dt = 0.8
    currentvalue={"prefix": "dt: "},
    pad={"t": 50},
    steps=steps
)]

# Update the layout to include slider
fig.update_layout(
    sliders=sliders,
    title="Numerical Differentiation (dt = 0.8)",
    xaxis_title='t',
    yaxis_title="df/dt",
    showlegend=True,
    width=800,  # Set the width of the figure
    height=600   # Set the height of the figure
)

# Show the plot
fig.show()
```



Questions:

- What did you see different in the three methods?
- at which places they appear more different that the actual fit?
- what could be the reason?

4. Elaborate (20 minutes)

- **Objective:** understanding about the use of each method and their limitations.
- application for real life data.

Numerical errors

Truncation Error and Roundoff Error in Numerical Differentiation

When using numerical methods to approximate derivatives, two types of errors arise: **truncation error** and **roundoff error**. Understanding these errors is crucial for determining the accuracy and limitations of numerical differentiation.

1. Truncation Error

Truncation error occurs when an infinite series, such as a Taylor series, is approximated by a finite number of terms. In numerical differentiation, we often approximate the derivative of a function using finite difference methods, which are based on truncating the Taylor expansion at a specific order.

For example, the **forward difference approximation** for the first derivative of a function $f(x)$ is given by:

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

This formula is derived from the Taylor series expansion of $f(x+h)$ around x:

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f^{(3)}(x) + \dots$$

By ignoring the higher-order terms, we introduce a truncation error. The truncation error for the forward difference method is proportional to $O(h)$, which means that as h decreases, the error reduces linearly with h.

Similarly, for the **central difference approximation**:

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

The truncation error is proportional to $O(h^2)$, meaning that the error decreases quadratically as h becomes smaller, making the central difference method more accurate than the forward difference method.

2. Roundoff Error

Roundoff error arises due to the limited precision with which computers represent real numbers. In floating-point arithmetic, numbers are stored with a finite number of digits, and as a result, every arithmetic operation introduces a small error.

In numerical differentiation, roundoff error becomes significant when subtracting two nearly equal numbers, such as $f(x+h)$ and $f(x-h)$ for small h. This phenomenon is known as **catastrophic cancellation**. The difference between these two values is small, and the finite precision of floating-point numbers can lead to large relative errors.

The roundoff error is proportional to:

$$\frac{\epsilon_{\text{mach}}}{h}$$

where ϵ_{mach} is the machine epsilon, representing the smallest number that can be distinguished from 1 in floating-point arithmetic. As the step size h becomes smaller, the roundoff error increases because of the $1/h$ factor.

3. The Trade-off Between Truncation Error and Roundoff Error

In practice, there is a trade-off between truncation error and roundoff error when choosing the step size h in numerical differentiation. As h decreases:

- **Truncation error** decreases because the approximation becomes more accurate.
- **Roundoff error** increases due to the limited precision of floating-point arithmetic.

Thus, there is an **optimal step size** h_{opt} where the total error, which is the sum of truncation and roundoff errors, is minimized.

The total error $E(h)$ can be expressed as:

$$E(h) = \text{Truncation Error} + \text{Roundoff Error}$$

where:

$$E(h) \approx O(h^n) + O\left(\frac{\epsilon_{\text{mach}}}{h}\right)$$

Here, n represents the order of accuracy of the method (e.g., $n = 1$ for forward difference and $n = 2$ for central difference).

The goal in numerical differentiation is to choose h such that the total error is minimized by balancing these two sources of error.

Summary

- **Truncation error** results from approximating a function's derivative by ignoring higher-order terms in the Taylor series. The error is reduced by decreasing the step size h, but only up to a point.
- **Roundoff error** arises from the finite precision of computers and becomes significant when subtracting nearly equal numbers, especially for small h. The error increases as h becomes too small.
- The **optimal step size** is found by balancing truncation error and roundoff error, minimizing the total error in numerical differentiation.

Activity

- run the below code and try to find the answers

In [4]:
```python
import numpy as np
import matplotlib.pyplot as plt

# Define the function and its exact derivative
def f(x):
    return np.sin(x)

def f_prime_exact(x):
    return np.cos(x)

# Define the numerical differentiation methods
def forward_diff(f, x, h):
    return (f(x + h) - f(x)) / h

def backward_diff(f, x, h):
    return (f(x) - f(x - h)) / h

def central_diff(f, x, h):
    return (f(x + h) - f(x - h)) / (2 * h)

# Define machine epsilon for roundoff error estimation
epsilon_machine = np.finfo(float).eps

# Compute truncation error and roundoff error
def compute_errors(f, f_prime_exact, x, h_values):
    truncation_errors_forward = []
    truncation_errors_backward = []
    truncation_errors_central = []

    roundoff_errors_forward = []
    roundoff_errors_backward = []
    roundoff_errors_central = []

    # Exact value of derivative
    f_prime_exact_value = f_prime_exact(x)

    for h in h_values:
        # Forward difference method
        f_prime_forward = forward_diff(f, x, h)
        truncation_errors_forward.append(np.abs(f_prime_forward - f_prime_exact_value))
        roundoff_errors_forward.append(np.abs(epsilon_machine / h))

        # Backward difference method
        f_prime_backward = backward_diff(f, x, h)
```

```
        truncation_errors_backward.append(np.abs(f_prime_backward - f_prime_exact_value))
        roundoff_errors_backward.append(np.abs(epsilon_machine / h))

        # Central difference method
        f_prime_central = central_diff(f, x, h)
        truncation_errors_central.append(np.abs(f_prime_central - f_prime_exact_value))
        roundoff_errors_central.append(np.abs(epsilon_machine / h))

    return (truncation_errors_forward, truncation_errors_backward, truncation_errors_central,
            roundoff_errors_forward, roundoff_errors_backward, roundoff_errors_central)

# Set the point x at which to evaluate the derivative
x = 1.0

# Define a range of step sizes (h values)
h_values = np.logspace(-16, -1, 50)

# Compute the errors for each method
(truncation_errors_forward, truncation_errors_backward, truncation_errors_central,
 roundoff_errors_forward, roundoff_errors_backward, roundoff_errors_central) = compute_errors(f

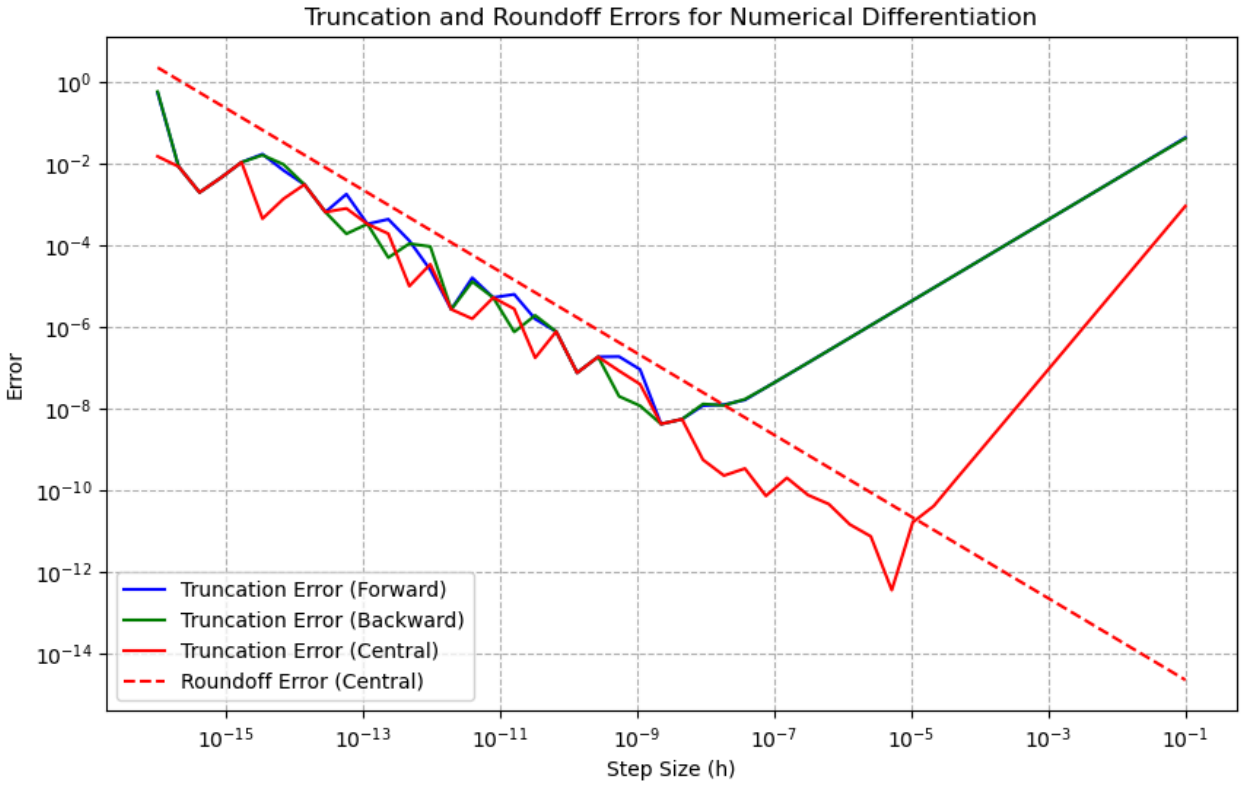
# Plot the truncation and roundoff errors
plt.figure(figsize=(10, 6))

# Forward difference method
plt.loglog(h_values, truncation_errors_forward, label='Truncation Error (Forward)', color='blue
#plt.loglog(h_values, roundoff_errors_forward, label='Roundoff Error (Forward)', linestyle='--'

# Backward difference method
plt.loglog(h_values, truncation_errors_backward, label='Truncation Error (Backward)', color='gr
#plt.loglog(h_values, roundoff_errors_backward, label='Roundoff Error (Backward)', linestyle='-

# Central difference method
plt.loglog(h_values, truncation_errors_central, label='Truncation Error (Central)', color='red'
plt.loglog(h_values, roundoff_errors_central, label='Roundoff Error (Central)', linestyle='--',

# Plot settings
plt.xlabel('Step Size (h)')
plt.ylabel('Error')
plt.title('Truncation and Roundoff Errors for Numerical Differentiation')
plt.legend()
plt.grid(True, which="both", ls="--")
plt.show()
```



Questions:

- 1. What is the trade-off between truncation error and roundoff error as the step size decreases?
  2. Why does the central difference method generally produce lower truncation error compared to the forward and backward difference methods?
  3. How can you determine the optimal step size h for numerical differentiation based on the error plot?
  4. some more questions

Application to real life data

In [5]:
```
# Load the data from the CSV file
data = pd.read_csv('sin_x_random_increment_data.csv')

# Extract x and y values from the DataFrame
x_vals = data['x'].values
y_vals = data['y'].values
dydx_E = np.cos(x_vals)

# Initialize an array to store the derivative values
dy_dx = np.zeros(len(x_vals))

# Forward difference at the beginning
dy_dx[0] = (y_vals[1] - y_vals[0]) / (x_vals[1] - x_vals[0])

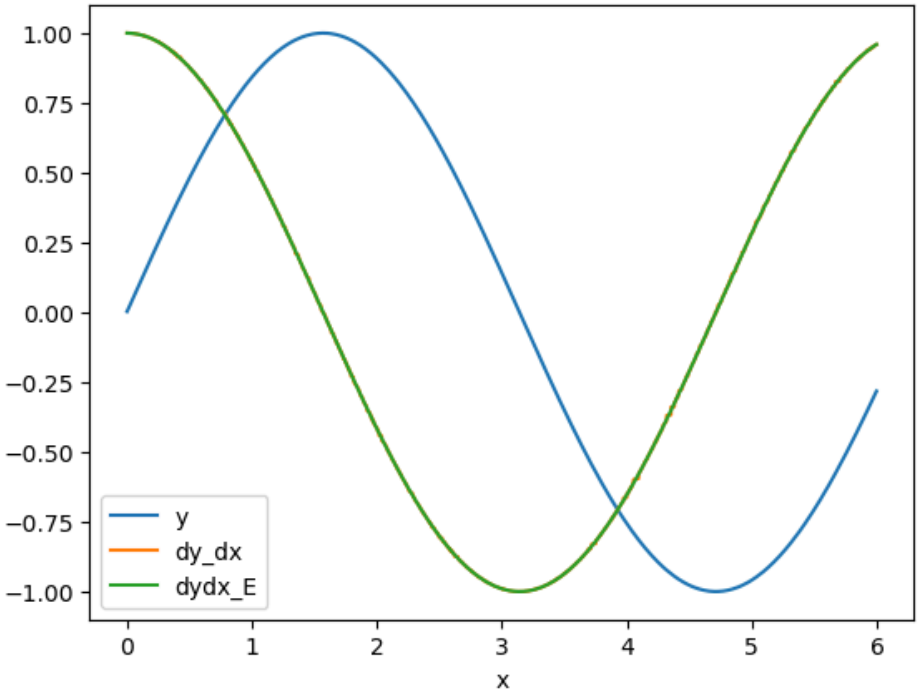
# Central difference for the middle points
for i in range(1, len(x_vals) - 1):
    dy_dx[i] = (y_vals[i + 1] - y_vals[i - 1]) / (x_vals[i + 1] - x_vals[i - 1])

# Backward difference at the end
dy_dx[-1] = (y_vals[-1] - y_vals[-2]) / (x_vals[-1] - x_vals[-2])

# Create a new DataFrame with x, y, and the derivative values
result_data = pd.DataFrame({
    'x': x_vals,
    'y': y_vals,
```

```
    'dy_dx': dy_dx,
    'dydx_E':dydx_E
})
result_data.plot('x',['y','dy_dx','dydx_E'])
```

Out[5]: <Axes: xlabel='x'>



Your job here

- read the file **'perturbed_function_data.csv'**, do the derivative and plot it here.

In [ ]:

In [ ]:

Possible homework problems

1. Derivation of Higher-Order Accuracy Formula:

- **Problem**: Derive the **fourth-order accurate** central difference formula for the first derivative. You have already seen the second-order accurate central difference formula: $f'(x) \approx \frac{f(x+h)-f(x-h)}{2h} + \mathcal{O}(h^2)$ Now, derive a formula for the central difference approximation that has an error term proportional to h^4.
- **Instructions**: Start by expanding $f(x+h)$ and $f(x-h)$ using the Taylor series up to the fifth derivative and solve for the higher-order terms to reduce the truncation error to h^4.

---

2. Plotting Truncation Error for Different Orders of Accuracy:

- **Problem**: Write code to compare the truncation errors of **second-order** and **fourth-order** accurate central difference formulas.
  - For the second-order accurate formula, use: $f'(x) \approx \frac{f(x+h)-f(x-h)}{2h}$
  - For the fourth-order accurate formula, use the expression you derived in **Problem 1**.
- **Instructions**:

1. Plot the truncation error for both the second-order and fourth-order formulas as a function of the step size h.
2. Use step sizes in the range $h = 10^{-5}$ to $h = 10^{-1}$.
3. On the same graph, compare the truncation errors of the two methods.
4. Describe how the errors change as h decreases, and explain why the fourth-order method has better accuracy at small values of h.

---

3. **Finding Maxima and Minima Using Numerical Methods:**

- **Problem**: Find the **maxima** and **minima** of the following equation using numerical methods:
  $f(x) = x^4 - 4x^2 + 3$
  - Write a Python program to find the critical points of the function.
  - Use numerical differentiation (central difference) to approximate the first and second derivatives of the function and classify the critical points as maxima or minima.
- **Questions**:
  1. Does your code accurately calculate the maxima or minima?
  2. If it doesn't, explain the potential reasons for the inaccuracy (e.g., step size, truncation error, etc.).
  3. What steps could you take to improve the accuracy of your code?

---

4. **Better Methods for Finding Maxima and Minima:**

- **Problem**: In the previous problem, you used numerical differentiation to find the maxima and minima of the function. Now, explore and suggest a better method for calculating these extrema.
- **Questions**:
  1. What other methods could you use to find maxima and minima more reliably?
  2. Write code using an optimization method (e.g., `scipy.optimize.minimize_scalar` for minima or gradient ascent for maxima) to find the extrema.
  3. Compare the results from this method with your previous numerical differentiation method.
  4. Can you use any other method to calculate it accurately?

Submission Instructions:

- For each problem, submit your derivations, code, and plots as part of a report.
- Ensure your code is well-commented and that you explain any decisions you made about parameters like step size or method selection.
- For the plots, include proper labels, titles, and legends to ensure clarity.

Post-Assessment

- Please take the pre-assessment questions and try to answer at best of your knowledge
- choose directly I don't know if you dont understand the question.
- **Click in the below link to start the assessment**
- https://forms.office.com/Pages/ResponsePage.aspx?id=o-nf9HjKmEm3p-bQaS6s5vHw702fgfxLigWNapFeBOxUMTg3REo0RVhWWUVLTlAxQTY1TVg0Q1ZHVy4u
- **Check student's response online**
- https://forms.office.com/Pages/DesignPageV2.aspx?origin=NeoPortalPage&subpage=design&id=o-nf9HjKmEm3p-bQaS6s5vHw702fgfxLigWNapFeBOxUN0RGTEk0MFc0MFQ4Sk5IWURQWEZXSTBWNC4u&analysis=false&top

Class assessment survey

- Please take this survey at your convenience
- https://forms.office.com/Pages/ResponsePage.aspx?id=o-nf9HjKmEm3p-bQaS6s5vHw702fgfxLigWNapFeBOxUQ0NFQjZFRzBZUFYwMFFWVTE1N1ZBWEZSUy4u

Bonus topic

In [6]:
```
# Define the function f(x) = x^2 + 4x + 4
def f(x):
    return x**2 + 4*x + 4

# Define the first derivative of the function f'(x) = 2x + 4
def f_prime(x):
    return 2*x + 4

##############################################################################################
# Gradient Descent Algorithm
def gradient_descent(learning_rate=0.001, max_iters=100, tolerance=1e-6):
    x = 5.0  # Initial guess for x
    history = [x]  # Store the path of x values
    for i in range(max_iters):
        gradient = f_prime(x)
        x_new = x - learning_rate * gradient
        if abs(x_new - x) < tolerance:
            break
        x = x_new
        history.append(x)
    return x, history

# Run gradient descent
final_x, x_history = gradient_descent(learning_rate=0.1)

# Generate x values and f(x) values for the plot
x_vals = np.linspace(-6, 6, 400)
y_vals = f(x_vals)
##############################################################################################
##  ANIMATION PART
##############################################################################################
# Create the initial figure
fig = go.Figure()

# Add the static function plot (f(x)) as the first trace
static_trace = go.Scatter(
    x=x_vals,
    y=y_vals,
```

```
    mode='lines',
    name='f(x) = x^2 + 4x + 4',
    line=dict(color='blue')
)
fig.add_trace(static_trace)

# Initialize the gradient descent point
gd_point = go.Scatter(
    x=[x_history[0]],
    y=[f(x_history[0])],
    mode='markers',
    marker=dict(color='red', size=10),
    name='Gradient Descent Path'
)
fig.add_trace(gd_point)

# Create frames for the animation
frames = []
for i in range(1, len(x_history)):
    frames.append(go.Frame(
        data=[
            # Include the static function plot in each frame
            static_trace,
            # Update the gradient descent point
            go.Scatter(
                x=[x_history[i]],
                y=[f(x_history[i])],
                mode='markers',
                marker=dict(color='red', size=10),
                name='Gradient Descent Path'
            )
        ],
        name=f'frame{i}'
    ))

# Add the frames to the figure
fig.frames = frames

# Define animation settings
animation_settings = dict(
    frame=dict(duration=500, redraw=True),
    fromcurrent=True,
    transition=dict(duration=300, easing='linear')
)

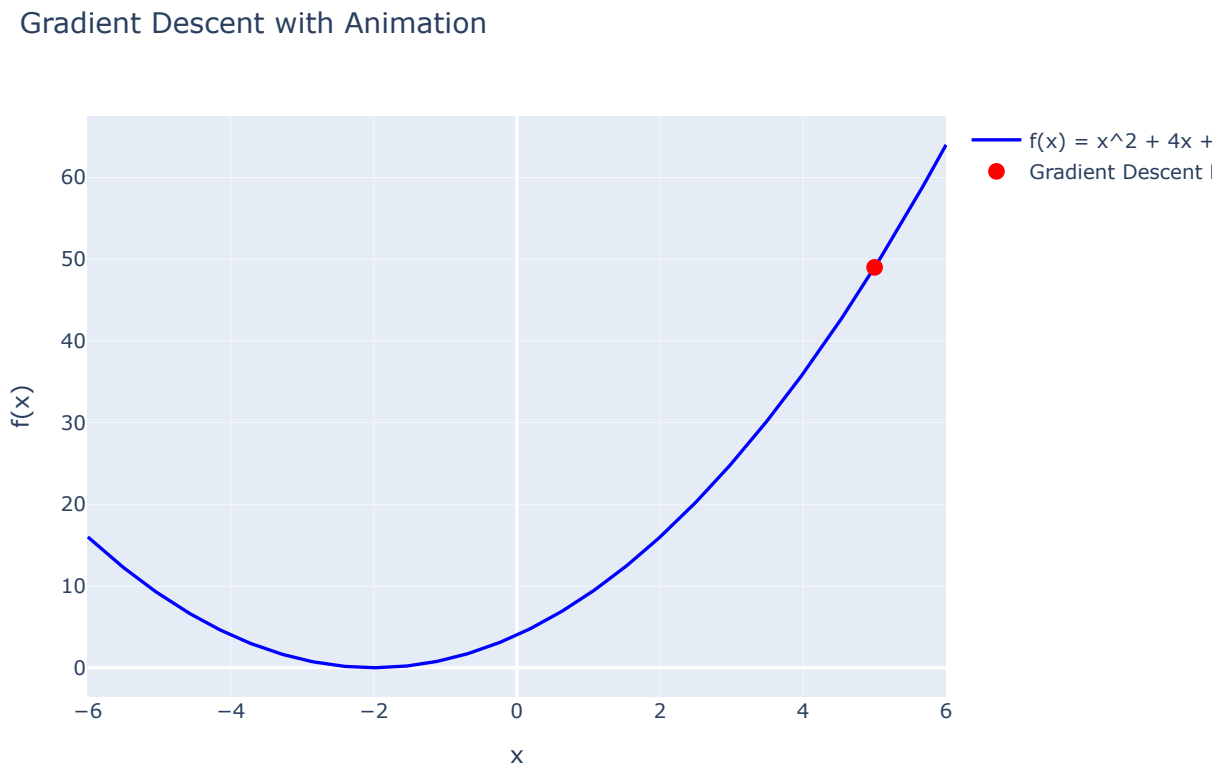
# Add play/pause buttons
fig.update_layout(
    updatemenus=[
        {
            'type': 'buttons',
            'buttons': [
                {
                    'label': 'Play',
                    'method': 'animate',
                    'args': [None, animation_settings]
                },
                {
                    'label': 'Pause',
                    'method': 'animate',
                    'args': [
                        [None],
                        dict(frame=dict(duration=0, redraw=False),
                             mode='immediate',
                             transition=dict(duration=0))
                    ]
                }
            ],
            'showactive': False,
```

```
            'direction': 'left',
            'x': 0.1,
            'y': -0.1,
            'pad': {'r': 10, 't': 70},
            'xanchor': 'right',
            'yanchor': 'top'
        }
    ]
)

# Add axis labels, title, and grid
fig.update_layout(
    xaxis_title='x',
    yaxis_title='f(x)',
    title='Gradient Descent with Animation',
    showlegend=True,
    width=800,  # Set the width of the figure
    height=600   # Set the height of the figure
)

# Display the plot
fig.show()

# Print the final value of x
print(f"Final value of x: {final_x}, f(x) = {f(final_x)}")
```




```
Final value of x: -1.9999956060287851, f(x) = 1.9306778398231472e-11
```

In [7]:
```
##
```