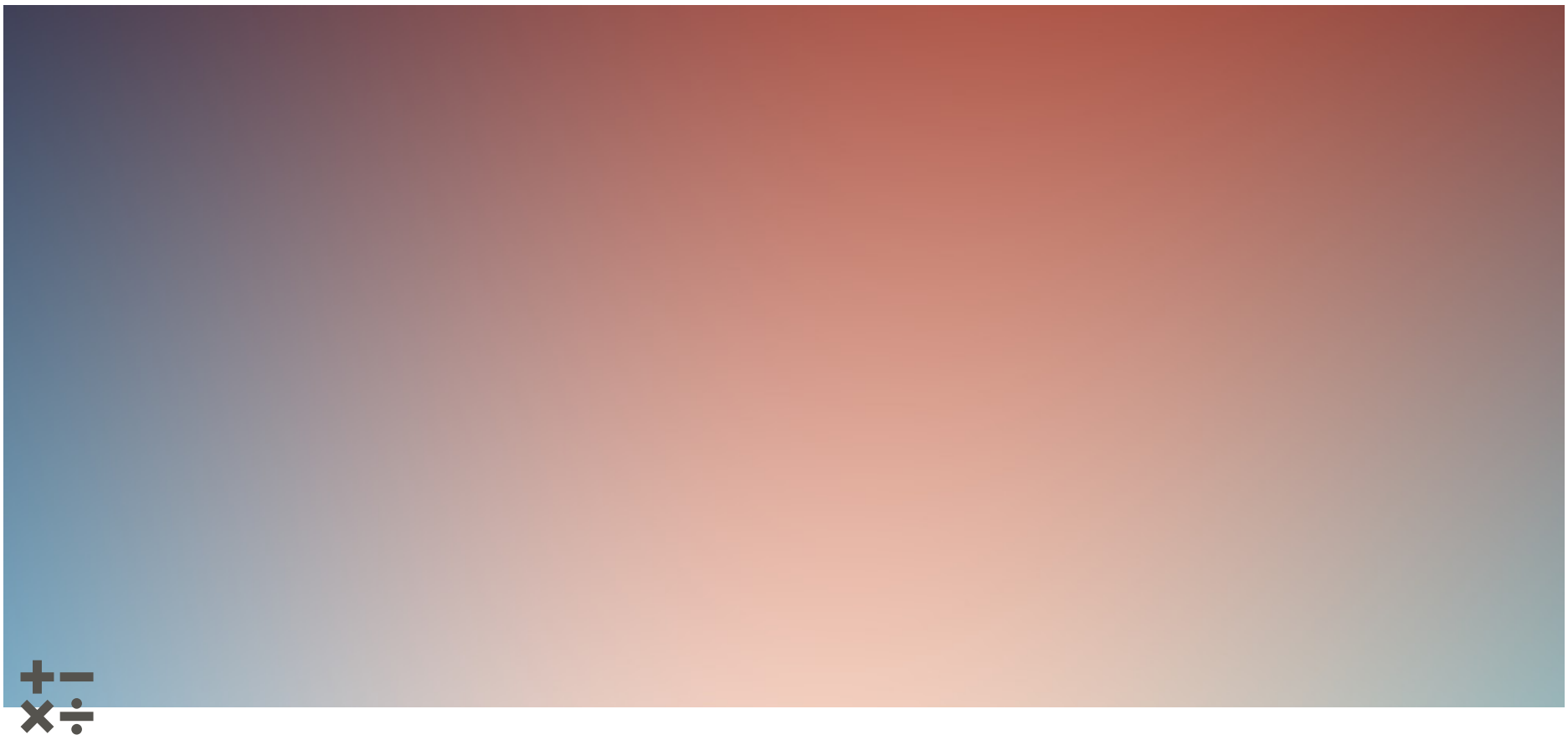


Analyzing Floating Point Errors and Finite Difference Methods

Introduction

Floating point arithmetic is essential in computer science and numerical analysis for representing and manipulating real numbers. However, its limitations can lead to various errors and inaccuracies in calculations. This section explores these errors and their implications for computational accuracy.

Before diving into specific floating point errors, it's important to understand finite difference methods. These numerical techniques approximate function derivatives and are widely used in fields like physics, engineering, and finance. Finite difference methods replace continuous derivatives with discrete approximations based on nearby function values.

While useful for complex functions or when analytical solutions are difficult, the accuracy of finite difference methods can be compromised by the same floating point errors discussed in this section. Understanding both floating point arithmetic and finite difference methods is crucial for developing reliable numerical algorithms and interpreting results accurately

Floating Point Arithmetic Errors

Inaccuracy Due to Finite Precision

Computers represent numbers using a limited number of bits, which leads to **finite precision** and can cause inaccuracies. In **floating-point representation**, numbers are stored with a mantissa and exponent, but not all real numbers can be exactly represented. This results in **rounding errors**, where numbers are approximated, especially when adding or subtracting values of different magnitudes.

Rounding, underflow, and overflow are common issues. **Round-off errors** occur when numbers are rounded to fit within precision limits, while **underflow** causes very small numbers to be represented as zero and **overflow** happens when numbers exceed the representable range.

Finite precision can even violate basic properties like the **associative property** of addition, where:

$$(a + b) + c = a + (b + c)$$

Rounding Errors

Here's a list of examples that demonstrate rounding errors due to finite precision:

Repeating Decimals

```
repeating_decimal = 1 / 3
print(f"Example 2: 1/3 = {repeating_decima
l:.30f}")  # Show 30 decimal places
# Example 2: 1/3 = 0.333333333333333314829616256247
```

After first 15 places float point number is completely inaccurate.

Addition/Subtraction Precision Loss

```
result_3 = 1.0000001 - 1.0000000
print(f"Example 3: 1.0000001 - 1.0000000 =
{result_3}")
# Example 3: 1.0000001 - 1.0000000 = 1.0000000005838672e-07
```

Result should've been `1e-7`.

Addition of Small and Large Numbers

```
large_number = 1.0
small_number = 1e-16
result_1 = large_number + small_number
print(f"Example 1: {large_number} + {small_
number} = {result_1}")
# Example 1: 1.0 + 1e-16 = 1.0
```

Completely disregards smaller number.

Accumulation of Small Rounding Errors

```
a = 1e16
b = -1e16
c = 1.0
result1 = (a + b) + c
result2 = a + (b + c)
print(f"Example 4: ((a + b) + c = {result
1})")
print(f"Example 4: (a + (b + c) = {result
2})")
# Example 4: ((a + b) + c = 1.0)
# Example 4: (a + (b + c) = 0.0)
```

Getting different answers completely.

Underflow Errors

Underflow errors occur in floating-point arithmetic when a calculation results in a number that is smaller than the smallest representable value (the "denormalized" range) for that data type, causing it to be approximated as zero. Here are some examples in Python that illustrate underflow errors:

Subtracting Very Small Numbers

```
a = 1e-10
b = 1e-20
result = a - b
print(f"Example 1: {a} - {b} = {result:.20
f}\n")
```

Multiplying Small Numbers

```
a = 1e-200
b = 1e-200
result = a * b
print(f"Example 2: {a} * {b} = {result:.200
0f}\n")
```

```
# Example 1: 1e-10 - 1e-20 = 0.000000000099
99999999
```

Completely wrong result value.

Adding Large and Small Numbers

```
large_number = 1e10
small_number = 1e-10
result_4 = large_number + small_number
print(f"Example 3: {large_number} + {small_
number} = {result_4:.20f}\n")
# Example 3: 10000000000.0 + 1e-10 = 100000
00000.00000000000000000000
```

Disregarding smaller number altogether.

```
# Example 2: 1e-200 * 1e-200 = 0.0000000000
0000000000...000
```

Getting result of 0 instead of `1e-400`.

Exponential Function with Large Negative Input

```
import math

small_value = -1000  # A very small value f
or exp
result_5 = math.exp(small_value)
print(f"Example 4: exp({small_value}) = {re
sult_5:.20f}")  # This will be close to 0
# Example 4: exp(-1000) = 0.000000000000000
00000...000
```

Result is 0.

Overflow Errors

An overflow error occurs when a calculation produces a value that exceeds the maximum representable value for a data type.

Adding Large Numbers

```
a = 1e+308
b = 1e+308
result_1 = a + b
print(f"Example 1: {a} + {b} = {result_1}
\n")
# Example 1: 1e+308 + 1e+308 = inf
```

Result is inf i.e. infinity.

Multiplying Large Numbers

```
a = 1e+308
b = 1e+308
result_2 = a * b
print(f"Example 2: {a} * {b} = {result_2}
\n")
# Example 2: 1e+308 * 1e+308 = inf
```

Results in infinity.

Finite Differences

Overview of Finite Difference Methods

Finite difference methods are numerical techniques used to approximate derivatives of functions. They replace continuous derivatives with discrete approximations based on function values at nearby points. Common types include forward, backward, and central differences, each with its own advantages and limitations in terms of accuracy and stability.

Finite Difference Formulas with Different Precision

These formulas vary in their level of accuracy and computational complexity. Low order precision formulas are simpler and faster to compute but may introduce larger errors, especially for functions with rapid changes. High order precision formulas, while more computationally intensive, often provide more accurate approximations of derivatives, particularly for smooth functions.

The choice between different finite difference formulas depends on the specific problem, desired accuracy, and available computational resources. Let's examine some common formulas in more detail:

Low Order Precision Formulas

- Forward Difference

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

High Order Precision Formulas

- Central Difference

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

- Backward Difference

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

```
def forward_difference(x, h):
    return (f(x + h) - f(x)) / h


def backward_difference(x, h):
    return (f(x) - f(x - h)) / h
```

- Richardson Extrapolation

$$f'(x) \approx \frac{4 \cdot A(h) - A(2h)}{4 - 1}$$

$$A(h) = \frac{f(x+h) - f(x-h)}{2h}$$

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


def richardsons_extrapolation(x, h):
    return (4 * central_difference(x, h
/ 2) - central_difference(x, h)) / 3
```

Visualization of Errors

To better understand the errors associated with finite difference methods and floating point arithmetic, it's crucial to visualize these errors graphically. Visual representations can provide intuitive insights into the magnitude and behavior of errors across different scenarios. Let's examine how these errors manifest in both polynomial and trigonometric functions, starting with polynomial functions.

Polynomial Function

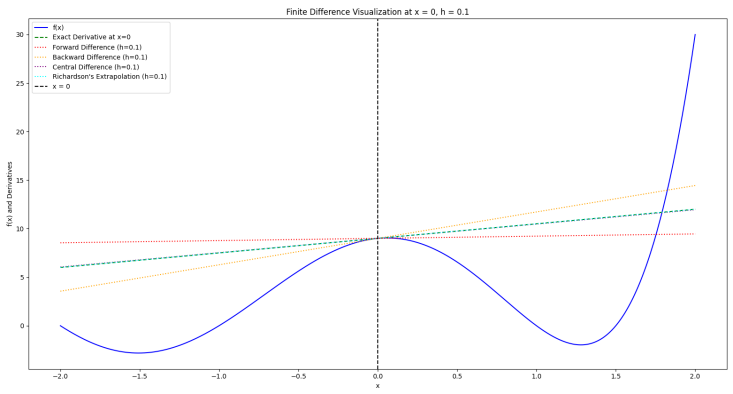
$$f(x) = x^5 + 3.5x^4 - 2.5x^3 - 12.5x^2 + 1.5x + 9$$

$$f'(x) = 5x^4 + 14x^3 - 7.5x^2 - 25x + 1.5$$

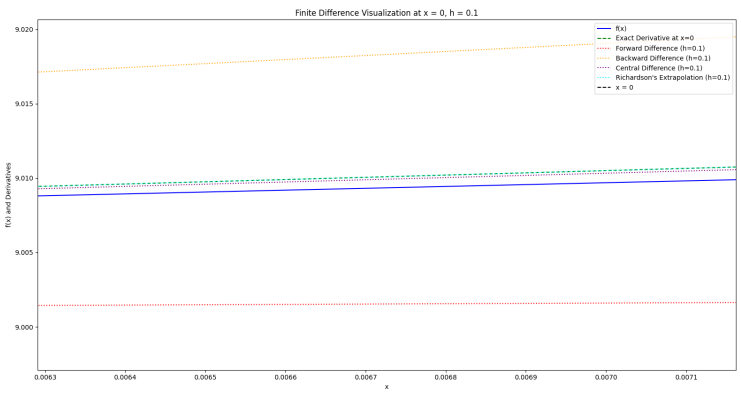
```
def f(x):
    return x**5 + 3.5 * x**4 - 2.5 * x**3 -
12.5 * x**2 + 1.5 * x + 9

def df(x):
    return 5 * x**4 + 14 * x**3 - 7.5 * x**
2 - 25 * x + 1.5
```

Tangent Line

Graph of tangent line at `x = 0`



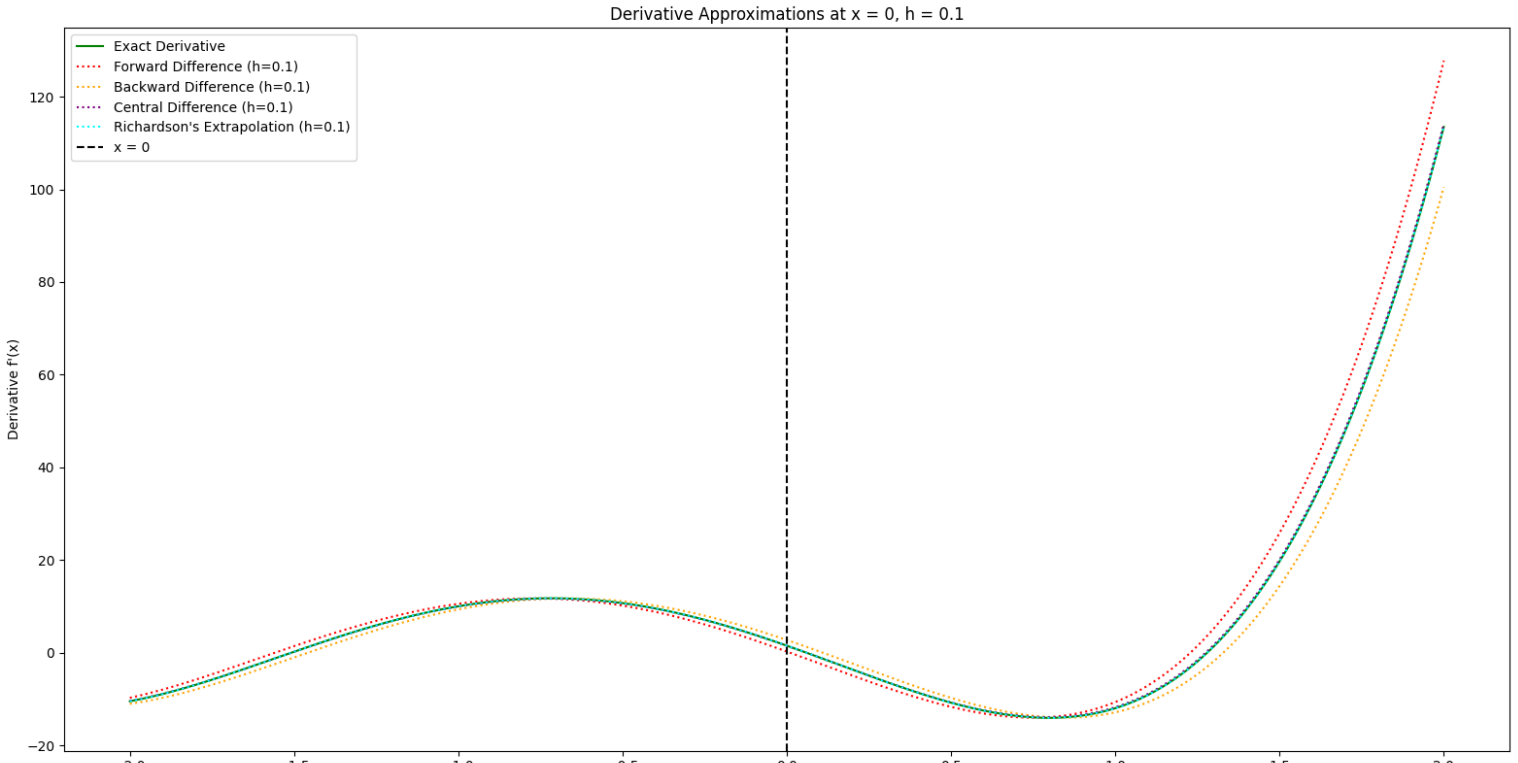
Some of lines may not be well visible and here is zoomed version.



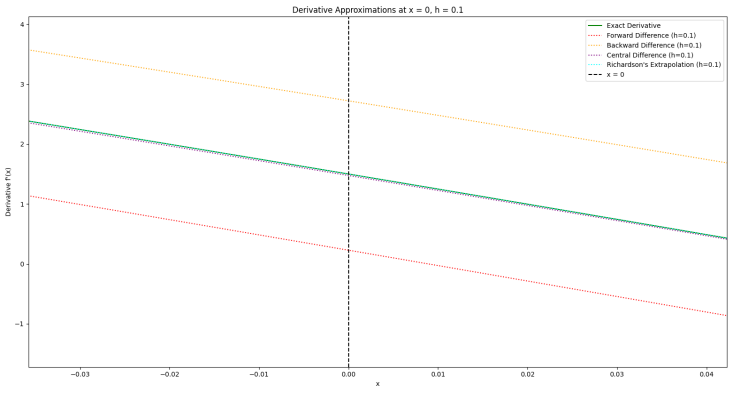
As expected first degree differences estimated worse than higher degree ones. Richardson’s method came closest which sits almost on the actual derivative while central difference is little bit off.

Derivative Graph

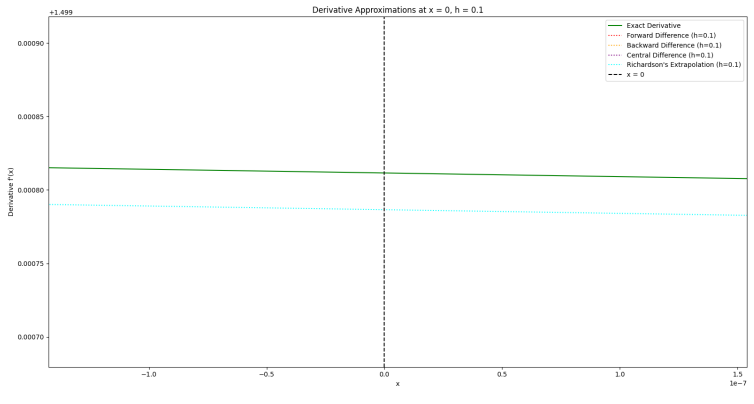
Following is a graph of 5 derivatives: actual derivative and then differences.



Zoomed in. Forward and central difference are way off than others, then comes central one and Richardson's

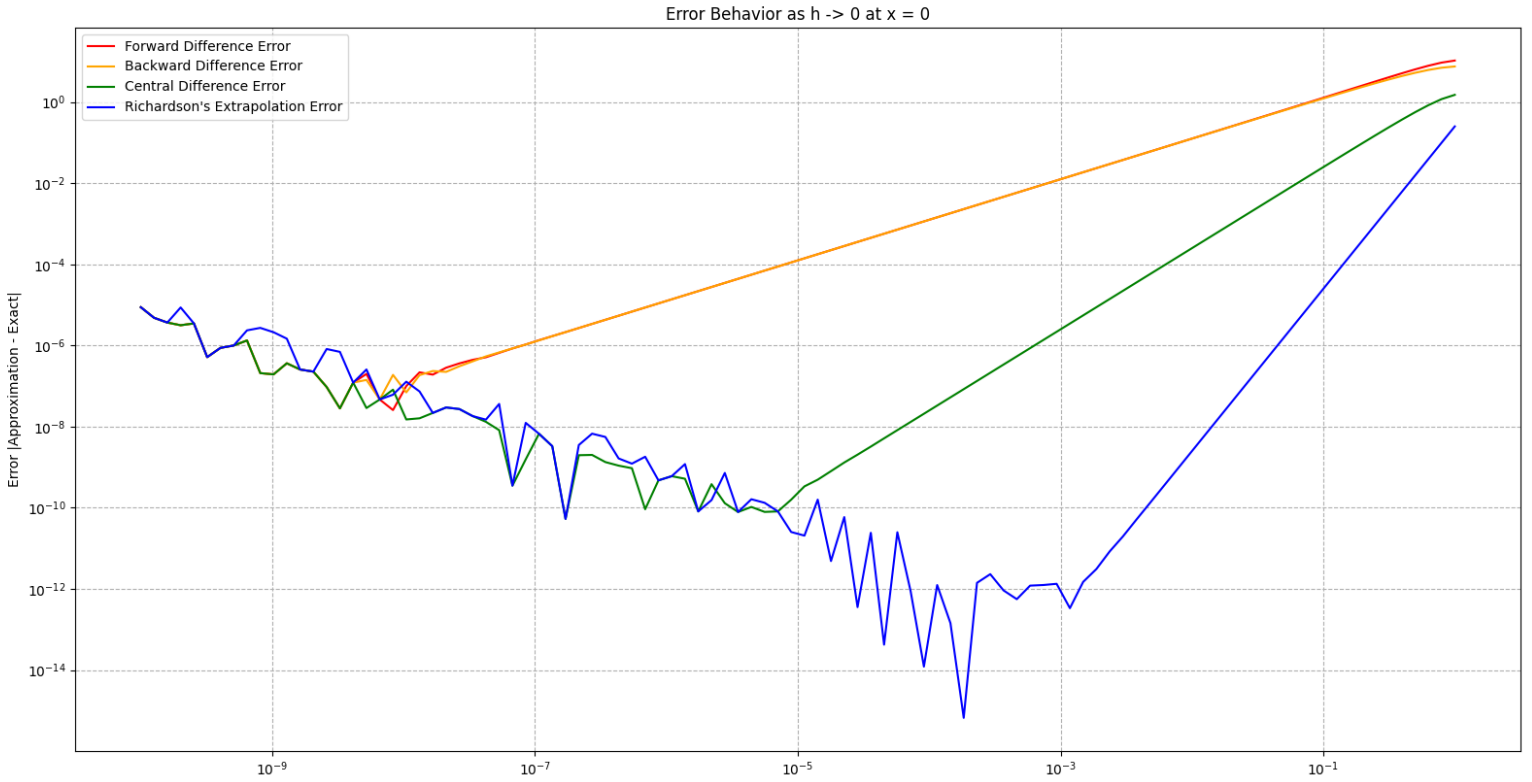


Forward and central difference are way off than others, then comes central one and Richardson's extrapolation almost is same line as actual derivative. Difference is in degree of 10 to -5 power.



Error Graph

Here is a logarithmic graph of errors when discretization term goes to 0 i.e. $h \to 0$



As $h \to 0$, error behavior shows following:

- `Forward` and `Backward` Differences: Both have increased errors, performing poorly as they are only first-order accurate.
- `Central` Difference: Performs better with smaller error due to its second-order accuracy, but still grows as ***h*** decreases.
- `Richardson's` Extrapolation: Has smallest error and best performance due to higher-order accuracy, but shows instability at very small ***h*** due to numerical precision limits.

Note that smallest ***h*** does not guarantee the best result due to **round-off errors**. As ***h*** becomes extremely small, subtractive cancellation occurs, increasing numerical errors.

Trigonometric Function

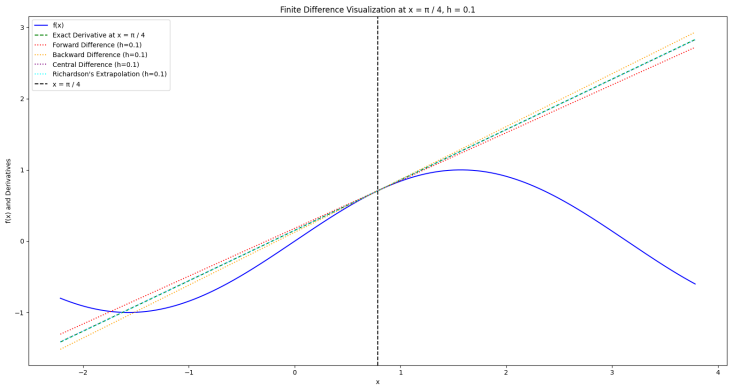
$$f(x) = sin(x)$$

$$f'(x) = \cos(x)$$

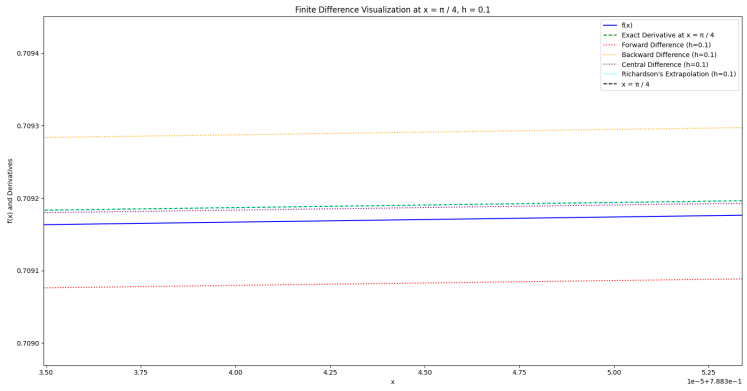
```
import math

def f(x):
    return math.sin(x)

def df(x):
    return math.cos(x)
```

Tangent Line

Graph of tangent line at $x = \pi / 4$.

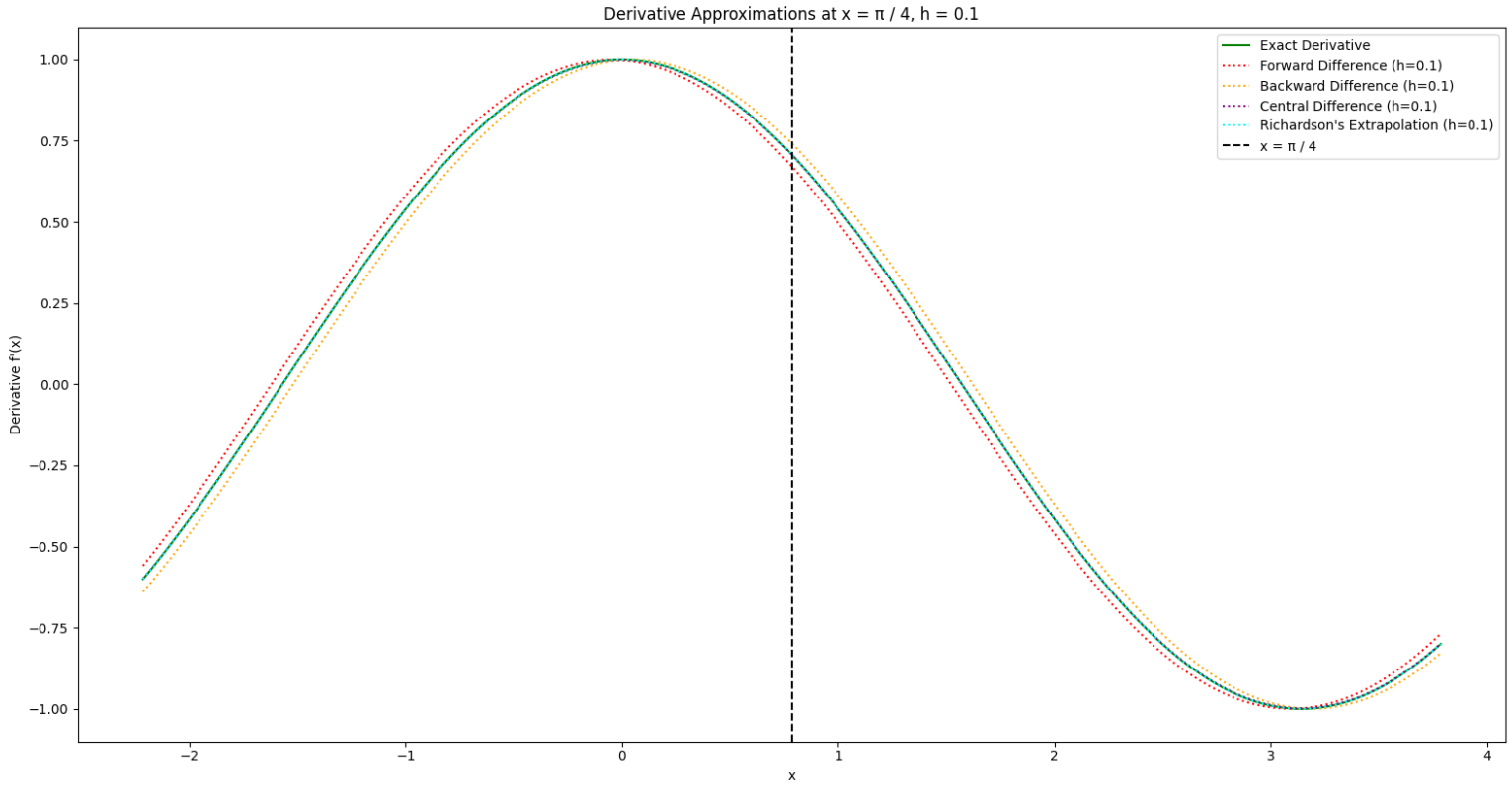


Zoomed in version.

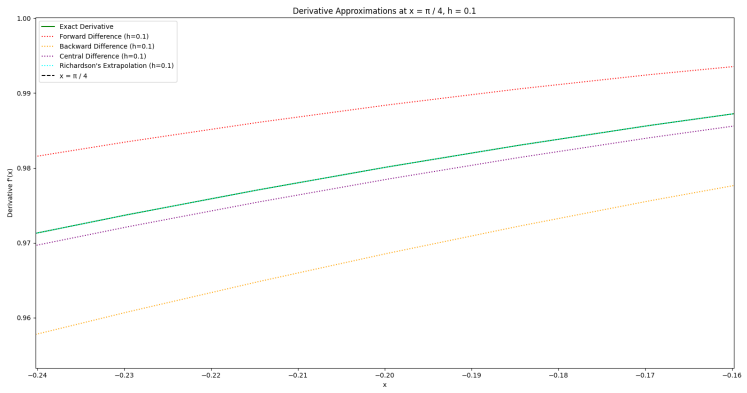


Derivative Graph

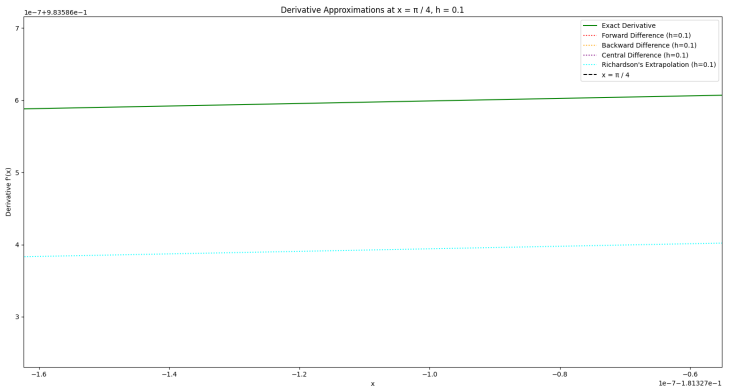
Here is a graph of differences as well as graph of actual derivative.



Zoomed in version.

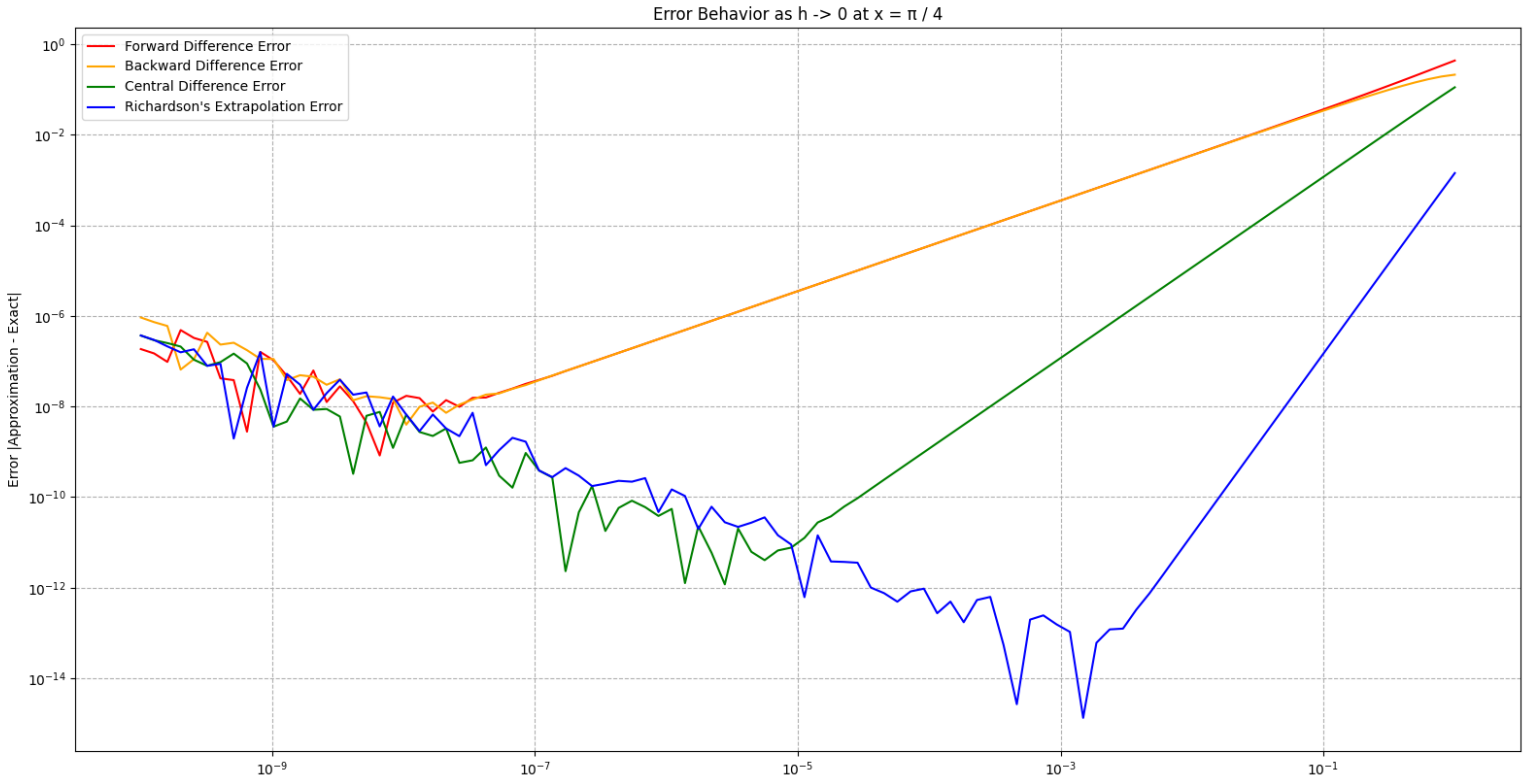


Difference between Richardson’s Extrapolation and actual derivative.



Error Graph

Here is a logarithmic graph of errors when discretization term goes to 0 i.e. $h \to 0$



As `h → 0`, error behavior shows following:

- `Forward` and `Backward` Differences: Both show increasing errors and perform poorly since they are only first-order accurate.
- `Central` Difference: Performs better with smaller errors due to its second-order accuracy but still experiences increasing error as ***h*** decreases.
- `Richardson's` Extrapolation: Achieves the smallest error and performs best due to its higher-order accuracy, but exhibits instability at very small ***h*** values due to numerical precision limits.

It's important to note that a smaller ***h*** does not always yield better results because of **round-off errors**. As ***h*** becomes extremely small, subtractive cancellation occurs, which leads to increasing numerical errors.

Conclusion

This analysis of floating point errors and finite difference methods highlights several key points:

- Floating point arithmetic can lead to inaccuracies due to finite precision in computer representations.
- Finite difference methods are crucial numerical techniques for approximating derivatives, with varying levels of precision and computational complexity.
- Low-order methods (forward and backward differences) are simpler but less accurate, while high-order methods (central difference and Richardson's extrapolation) offer improved accuracy at higher computational cost.
- Visualization of errors reveals that Richardson's extrapolation generally performs best, followed by central difference, with forward and backward differences showing larger errors.
- As the step size ***h*** approaches zero, numerical errors can increase due to round-off effects, demonstrating that smaller step sizes don't always yield better results.
- The choice of finite difference method depends on the specific problem, required accuracy, and available computational resources.

Understanding these concepts is crucial for effectively implementing numerical methods in scientific computing and engineering applications.