Chapter 4

Roundoff and Truncation Errors

Chapter Objectives

- Learning how to quantify error.
- Learning how error estimates can be used to decide when to terminate an iterative calculation.
- Understanding how roundoff errors occur because digital computers have a limited ability to represent numbers.

Objectives (cont)

- Knowing how to use the Taylor series to estimate truncation errors.
- Understanding how to write forward, backward, and centered finite-difference approximations of the first and second derivatives.
- Recognizing that efforts to minimize truncation errors can sometimes increase roundoff errors.

Error Definitions

- Absolute error ($|E_t|$): the absolute difference between the true value and the approximation.

$$E_t = true\ value - approximation$$

- True fractional relative error: the absolute true error divided by the true value.

$$true\ fractional\ relative\ error = \frac{true\ value - approximation}{true\ value}$$

- Relative error (ε_t): the true fractional relative error expressed as a percentage.

$$\varepsilon_t = \left| \frac{true\ value - approximation}{true\ value} \right| \times 100\%$$

Error Definitions (cont)

- The approximate percent relative error can be given as the approximate error divided by the approximation, expressed as a percentage

$$\varepsilon_a = \left| \frac{approximation\ error}{approximation} \right| \times 100\%$$

- For iterative processes, the error can be approximated as the difference in values between successive iterations.

$$\varepsilon_a = \left| \frac{present\ approximation - previous\ approximation}{present\ approximation} \right| \times 100\%$$

Using Error Estimates

- Often, when performing calculations, we may not be concerned with the sign of the error but are interested in whether the absolute value of the percent relative error is lower than a pre-specified tolerance ε_s. For such cases, the computation is repeated until $|\varepsilon_a| < \varepsilon_s$

$$\varepsilon_s = \left(0.5 \times 10^{2-n}\right)\%$$

- This relationship is referred to as a *stopping criterion*.

Example

$$e^x \cong 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^n}{n!}$$

$$\varepsilon_s = \left(0.5 \times 10^{2-3}\right)\% = 0.05\%$$

$$e^x = 1 + x$$

$$e^{0.5} = 1 + 0.5 = 1.5$$

$$\varepsilon_t = \left|\frac{1.64872127070013 - 1.5}{1.64872127070013}\right| \times 100\% \cong 9.02\%$$

$$\varepsilon_a = \left|\frac{1.5 - 1}{1.5}\right| \times 100\% \cong 33.3\%$$

Terms	Result	ε_t, %	ε_a, %
1	1	39.3	
2	1.5	9.02	33.3
3	1.625	1.44	7.69
4	1.645833333	0.175	1.27
5	1.648437500	0.0172	0.158
6	1.648697917	0.00142	0.0158

Roundoff Errors

- *Roundoff errors* arise because digital computers cannot represent some quantities exactly. There are two major facets of roundoff errors involved in numerical calculations:
  - Digital computers have size and precision limits on their ability to represent numbers.
  - Certain numerical manipulations are highly sensitive to roundoff errors.

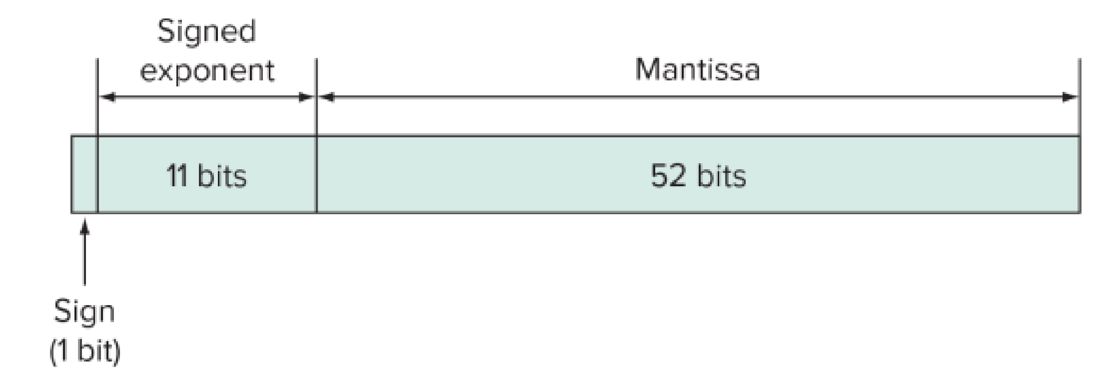
Computer Number Representation

$$(8 \times 10^3) + (6 \times 10^2) + (4 \times 10^1) + (2 \times 10^0) + (9 \times 10^{-1}) = 8642.9$$

$$(1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) + (1 \times 2^{-1}) = 101.1_2$$

$$4 + 0 + 1 + 0.5 = 5.5_{10}$$

$$\pm(1 + f) \times 2^e$$



$$Largest\ number = +1.111 \ldots 111 \times 2^{+1023}$$

$$Smallest\ number = +1.000 \ldots 000 \times 2^{-1022}$$

$$+2^{+1023} \cong 1.7977 \times 10^{308}$$

$$2^{-1022} \cong 2.2251 \times 10^{-308}$$

Hypothetical Computer

- Consider a hypothetical computer that stores numbers to 4 decimal places. Then on this computer
- $x1 = 0.12343 \rightarrow 0.1234$
- $x2 = 0.12344 \rightarrow 0.1234$
- $x2 - x1 = 0.00001 \rightarrow 0.0000$
- AND

Hypothetical Computer (Cont.)

- $y1 = 0.21354 \rightarrow 0.2135$
- $y2 = 0.21355 \rightarrow 0.2136$
- $y2 - y1 = 0.00001 \rightarrow 0.0001$
- Now Compute
- $\frac{y2-y1}{x2-x1} = 1 \rightarrow \infty$ (on the computer)
- **Main message**: care should be taken in computing with numbers with large decimal places

Roundoff Errors

- In Python the roundoff error could be taken as the computer epsilon, where $eps = 2.2204 \times 10^{-16}$ (in short format)
- *eps* is the smallest number that can be added to 1 to make the result different from 1

```
import numpy as np
eps = np.finfo(float).eps
print(eps)
```

- Check: write in command window

x=1; y=1+*eps*; x==y Gives 0

x=1; y=1+*eps/2*; x==y Gives 1

Roundoff Errors (Cont)

- A true number X is represented as

$X = \tilde{X} \pm e$, e being the roundoff error and $\hat{X}$ is the rounded off number.

For example, on a four-decimal place computer

$$0.13454 \rightarrow 0.1345 \rightarrow 0.1345 \pm 0.00005$$
$$0.13455 \rightarrow 0.1346 \rightarrow 0.1346 \pm 0.00005$$

Truncation Errors

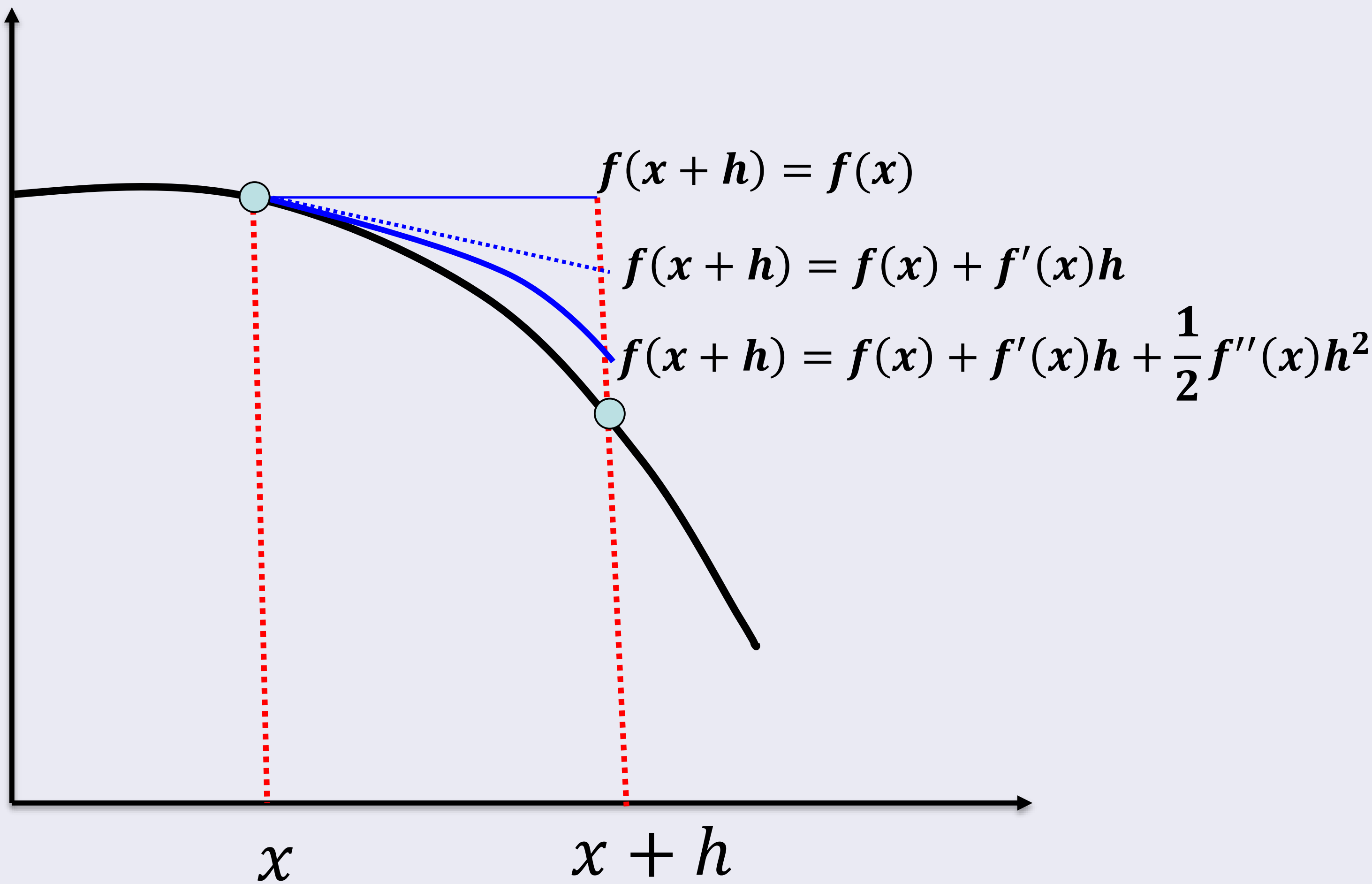
Truncation errors are those that result from using an approximation in place of an exact mathematical procedure.

The Taylor Theorem and Series

The *Taylor theorem* states that any smooth function can be approximated as a polynomial.

$$\boldsymbol{f(x+h)}$$
$$\boldsymbol{= f(x) + f'(x)h + \frac{1}{2!} f''(x)h^2 + \frac{1}{3!} f^{(3)}(x)h^3}$$
$$\boldsymbol{+ \cdots + \frac{f^{(n)}(x)}{n!} h^n + R_n}$$

Taylor Series



Truncation Error

- In general, the *n*th order Taylor series expansion will be exact for an *n*th order polynomial.

- In other cases, the remainder term R_n is of the order of h^{n+1}, meaning:

  - The more terms are used, the smaller the error, and

  - The smaller the spacing, the smaller the error for a given number of terms.

Taylor Series

$$f(x+h)$$
$$= f(x) + f'(x)h + \frac{1}{2!}f''(x)h^2 + \frac{1}{3!}f'''(x)h^3$$
$$+ \cdots.$$

$$f(x-h)$$
$$= f(x) - f'(x)h + \frac{1}{2!}f''(x)h^2 - \frac{1}{3!}f'''(x)h^3$$
$$+ \cdots.$$

Numerical Differentiation

- The first order Taylor series can be used to calculate approximations to derivatives:
  - Given: $\boldsymbol{f(x+h) = f(x) + f'(x)h + O(h^2)}$
  - Then: $\boldsymbol{f'(x) = \frac{f(x+h)-f(x)}{h} + O(h)}$

- This is termed a "forward" difference because it utilizes data at x and $x+h$ to estimate the derivative.

Differentiation (cont)

- There are also backward and centered difference approximations, depending on the points used:

- Backward:

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

- Centered:

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

Differentiation (cont)

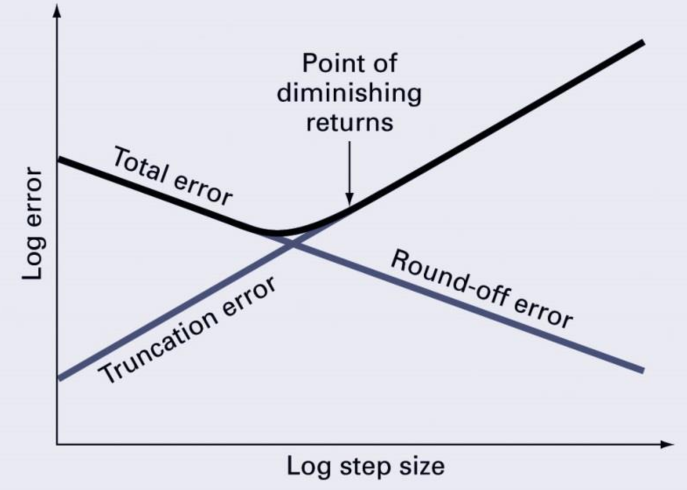
Second derivative

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

Total Numerical Error

- The *total numerical error* is the summation of the truncation and roundoff errors.

- The truncation error generally *increases* as the step size increases, while the roundoff error *decreases* as the step size increases - this leads to a point of diminishing returns for step size



Example: Total Error

Forward derivative

$$f'(x) = \frac{\tilde{f}(x+h) - \tilde{f}(x)}{h} + \frac{2e}{h} + \frac{M}{2}h$$

M is the maximum of the 2nd derivative

Hence $R = \frac{2e}{h} + \frac{M}{2}h$ is min for $h = 2\sqrt{\frac{e}{M}}$

Total Numerical Error

Centered Derivative

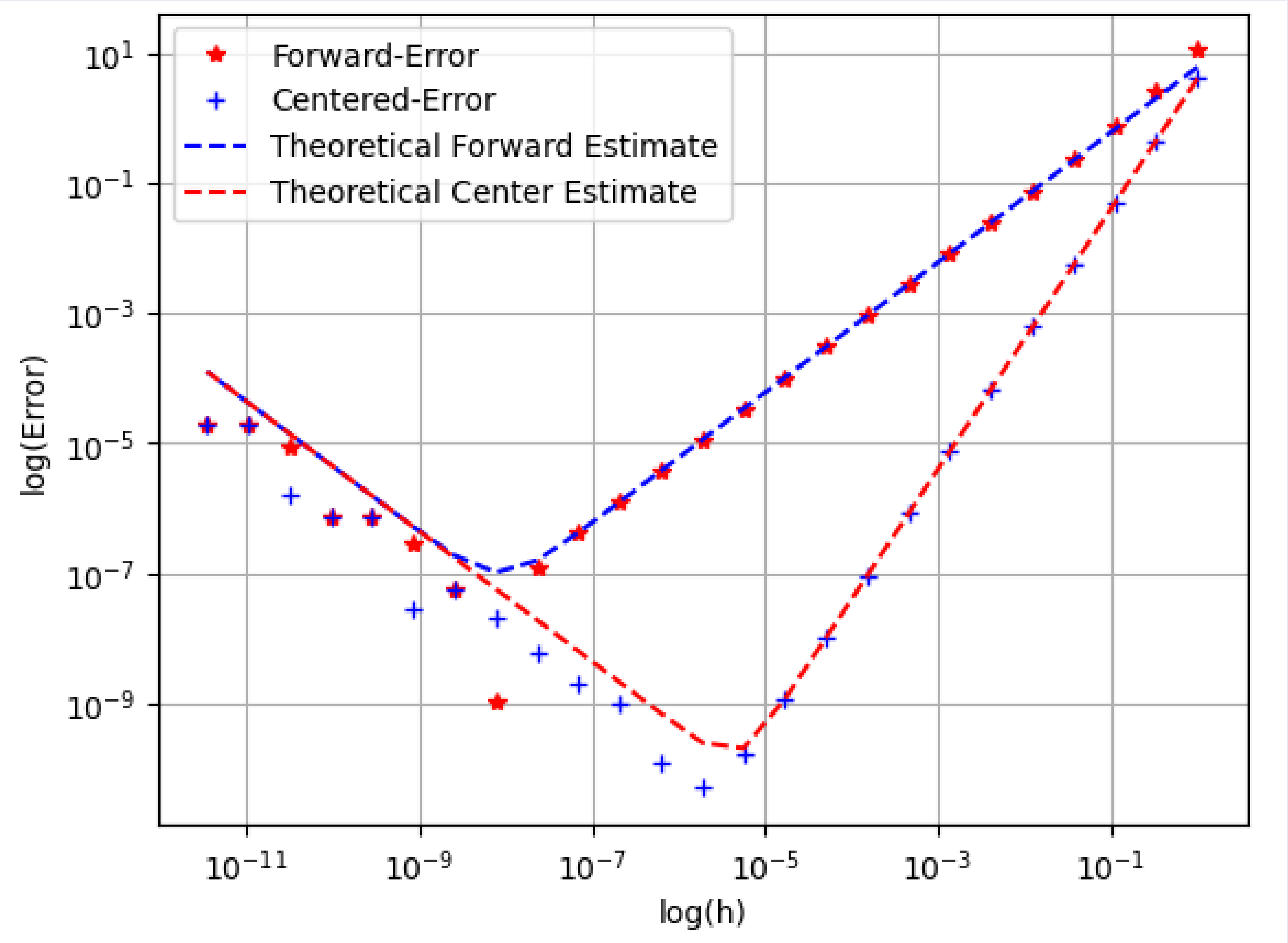
- Error in this case is $R = \frac{2e}{h} + \frac{M}{6}h^2$

- M is the max for 3rd derivative

- R is min for $h = \left(\frac{6e}{M}\right)^{\frac{1}{3}}$

```python
import numpy as np
import matplotlib.pyplot as plt

def comp_errors():
    # Zhoolideh 2024
    # To show Truncation and Roundoff Errors
    # Examples for 1st order derivatives: Forward and Central approx.

    np.set_printoptions(precision=14)  # Equivalent to Octave's `format long`

    n = 25
    x0 = 1
    h = 1.0

    def f(x):
        return x ** 4     # Define functions

    def fde(x):
        return 4 * x ** 3 # Analytical derivative

    # Initialize arrays for storing errors and step sizes
    Ef1 = np.zeros(n)  # Forward error
    Ec1 = np.zeros(n)  # Central error
    Rf1 = np.zeros(n)  # Theoretical forward error estimate
    Rc1 = np.zeros(n)  # Theoretical central error estimate
    H = np.zeros(n)    # Step sizes

    for k in range(n):
        # Forward and central difference approximations
        ffd1 = (f(x0 + h) - f(x0)) / h  # Forward derivative
        fcd1 = (f(x0 + h) - f(x0 - h)) / (2 * h)  # Central derivative

        # Calculate errors
        Ef1[k] = abs(ffd1 - fde(x0))  # Error in forward derivative
        Ec1[k] = abs(fcd1 - fde(x0))  # Error in central derivative

        # Roundoff errors (theoretical estimates)
        Rf1[k] = 2 * np.finfo(float).eps / h + 6 * h  # 6 results from f''(1)/2
        Rc1[k] = 2 * np.finfo(float).eps / h + 4 * h ** 2  # 4 results from f'''(1)/6

        # Store step size for plotting
        H[k] = h
        h /= 3  # Reduce step size

    # Plot the log-log graph
    plt.loglog(H, Ef1, 'r*', label='Forward-Error')
    plt.loglog(H, Ec1, 'o', label='Centered-Error')
    plt.loglog(H, Rf1, 'b-', label='Theoretical Forward Estimate')
    plt.loglog(H, Rc1, 'r-', label='Theoretical Center Estimate')

    plt.xlabel('log(h)')
    plt.ylabel('log(Error)')
    plt.legend(loc='best')
    plt.grid(True)
    plt.show()

    # Print the result in a table-like format
    print(f"{'h':>16} {'F-ERROR':>20} {'C-ERROR':>20}")
    for i in range(n):
        print(f"{H[i]:16.14f} {Ef1[i]:20.14f} {Ec1[i]:20.14f}")

# Call the function
comp_errors()
```

h	F-ERROR	C-ERROR
1.00000000000000	11.00000000000000	4.00000000000000
0.33333333333333	2.48148148148148	0.44444444444444
0.11111111111111	0.71742112482854	0.04938271604939
0.03703703703704	0.22775999593557	0.00548696844993
0.01234567901235	0.07468561891162	0.00060966316112
0.00411522633745	0.02475916806776	0.00006774035127
0.00137174211248	0.00823798196160	0.00000752670556
0.00045724737083	0.00274432062043	0.00000083630038
0.00015241579028	0.00091458766873	0.00000009292355
0.00005080526343	0.00030484191076	0.00000001032685
0.00001693508781	0.00010161169789	0.00000000115862
0.00000564502927	0.00003387037687	0.00000000016543
0**.00000188167642**	0.00001129003268	**0.00000000005091**
0.00000062722547	0.00000376388063	0.00000000012609
0.00000020907516	0.00000125323589	0.00000000102444
0.00000006969172	0.00000041210618	0.00000000208648
0.00000002323057	0.00000012535742	0.00000000606910
0.00000000774352	**0.00000000109962**	0.00000002040654
0.00000000258117	0.00000005844937	0.00000005844937
0.00000000086039	0.00000028564914	0.00000002757526
0.00000000028680	0.00000074664641	0.00000074664641
0.00000000009560	0.00000074664641	0.00000074664641
0.00000000003187	0.00000854401357	0.00000157601858
0.00000000001062	0.00001932796636	0.00001932796636
0.00000000000354	0.00001932796636	0.0000193279663



Forward-Error
Centered-Error
Theoretical Forward Estimate
Theoretical Center Estimate
10^{1}
10^{-1}
10^{-3}
10^{-5}
10^{-7}
10^{-9}
log(Error)
10^{-11}
10^{-9}
10^{-7}
10^{-5}
10^{-3}
10^{-1}
log(h)

Errors

In general it is not possible to carry out the error analysis as done here. However, the main message is to avoid unnecessarily small steps.

It is always advisable to carry out calculation for different values of steps and to check the stability of the outcome with these changes

Other Errors

- Blunders - errors caused by malfunctions of the computer or human imperfection.
- Model errors - errors resulting from incomplete mathematical models.
- Data uncertainty - errors resulting from the accuracy and/or precision of the data.

Assignment 1

Modify the 'comp_errors' code to calculate the analytic and numerical errors for the second derivative of the function x^4

Use $n = 20$ iterations for reduction of h

Submit your assignment through VC