Numerical Accuracy

Mark A. Austin

University of Maryland

austin@umd.edu
ENCE 201, Fall Semester 2023

July 6, 2023

Overview

Accuracy and Precision

Accuracy and Precision

Accuracy

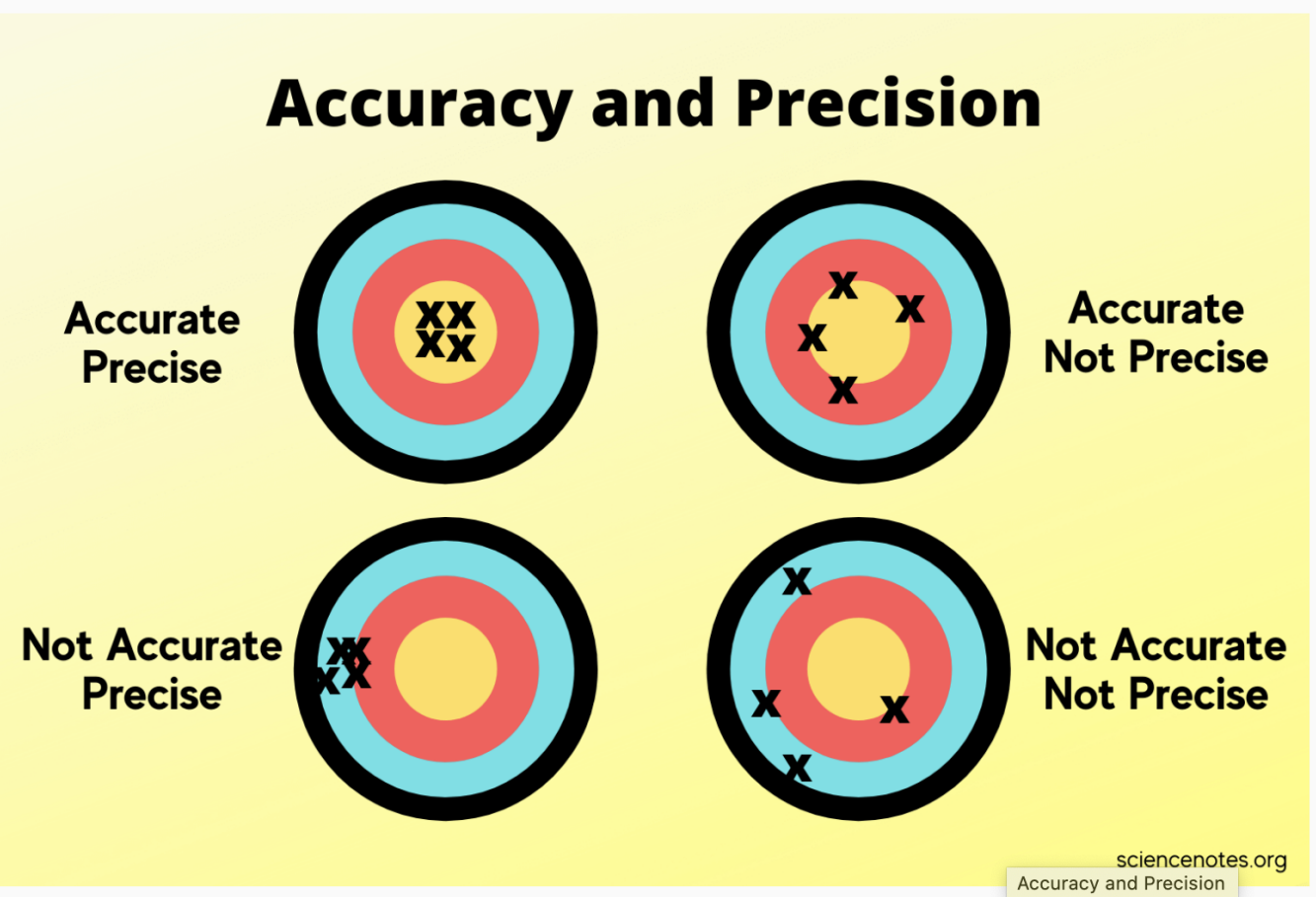
Accuracy refers to how close a measurement (or computational value) is to the true or accepted value.

Precision

Precision refers to how close measurements of the same item are to each other.

Accuracy and Precision

Variations in Accuracy and Precision:



Significant Digits of Accuracy

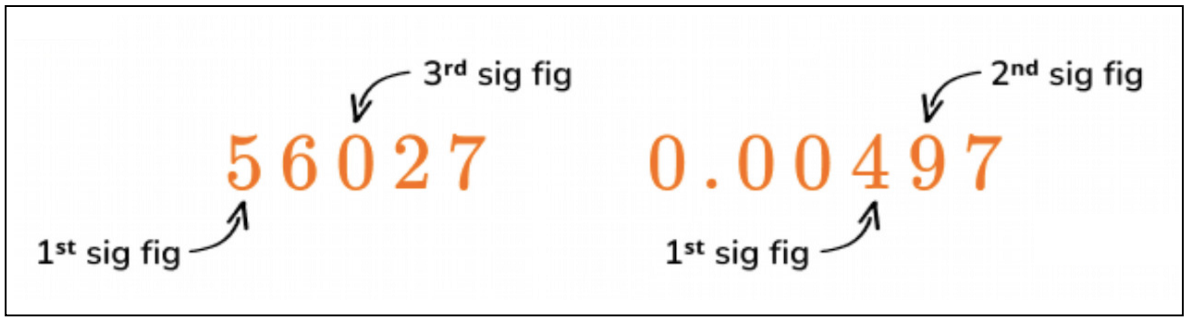
Significant Digits of Accuracy

Significant figures (digits of accuracy) show the precision of a numerical representation (or measurement).

Rules for determining significant figures:

- Non-zero digits are always significant.
- Any zeros located between non-zero digits are significant.
- Trailing zeros are only significant if a decimal point is present.

Examples:



Storage of Floating-Point Numbers

Definition. Floating point variables and constants are used represent values outside of the integer range (e.g., `3.4`, `-45.33` and `2.714`) and are either very large or small in magnitude, (e.g., `3.0e-25`, `4.5e+05`, and `2.34567890098e+19`).

IEEE 754 Floating-Point Standard. Specifies that a floating point number take the form:

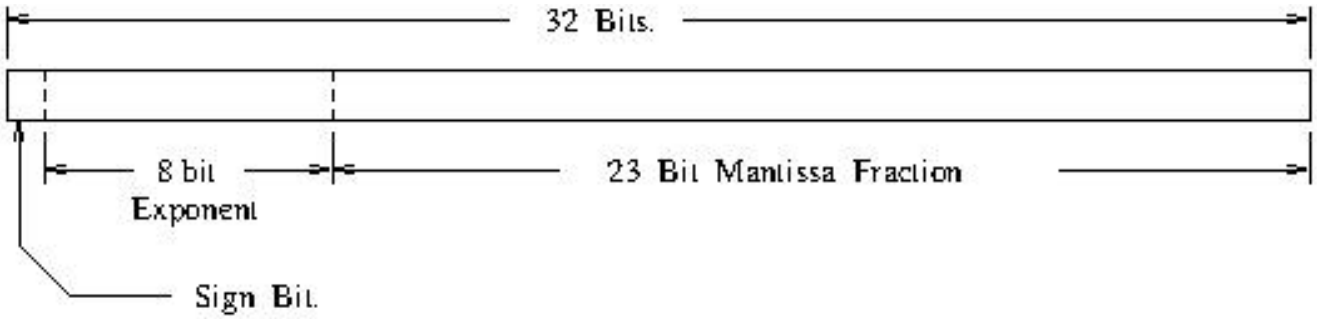
$$X = \sigma \cdot m \cdot 2^E. \tag{1}$$

Here:

- σ represents the sign of the number.
- m is the mantissa (interpreted as a fraction $0 < m < 1$).
- E is the exponent.

IEEE 754 Floating-Point Standard

Ensures floating point implementions and arithmetic are consistent across various types of computers (e.g., PC and Mac).



IEEE FLOATING POINT ARITHMETIC STANDARD FOR 32 BIT WORDS.

IEEE FLOATING POINT ARITHMETIC STANDARD FOR DOUBLE PRECISION FLOATS.

Largest and Smallest Floating-Point Numbers

```
=============================================================================
                    Default
Type    Contains      Value    Size       Range and Precision
=============================================================================

float   IEEE 754        0.0    32 bits    +- 13.40282347E+38 /
        floating point                     +- 11.40239846E-45

        Floating point numbers are represented to approximately
        6 to 7 decimal places of accuracy.

double  IEEE 754        0.0    64 bits    +- 11.7976931348623157E+308 /
        floating point                     +- 14.9406564584124654E-324

        Double precision numbers are represented to approximately
        15 to 16 decimal places of accuracy.
=============================================================================
```

Representation and Roundoff Errors

Representation Errors

Representation Errors

Representation errors occur when a number is represented with fewer digits than required for an exact value.

Representation errors can occur in two ways:

- The decimal representation of the number in base 10 has infinite digits, e.g., fraction $1/3$, π.

```
1/3 ---> 0.33333333333333 ... (base 10)
 pi ---> 3.14159265358979323846 ... (first 20 digits)
```

- The decimal representaion of the number in base 10 is has finite length, but an infinite number of digits when convered to base 2, e.g., 0.1.

```
0.1 (base 10) ---> 0.00011001100110011 ... (base 2)
```

Representation Errors

Example 1: Experiment with 32-bit floating point numbers.

```
fA = numpy.float32(4);          # <--- Create 32 bit float in Python.
fB = numpy.float32(3)
print("--- fA --> ", fA, " fB --> ", fB );
fC = fA/fB;                     # <--- Compute fC = fA/fB
print("--- Compute fC = (fA/fB) --> ", fC );
fD = (fC-1.0);                  # <--- Subtract 1 from fC ....
print("--- Compute fD = (fA/fB - 1) --> ", fD );
```

Output:

```
--- fA -->  4.0, fB -->  3.0
--- Compute fC = (fA/fB) -->  1.3333334
--- Compute fD = (fA/fB - 1) -->  0.3333333730697632
--- Compute 300*fD (answer should be 100) -->  100.00001192092896
--- Compute fSum = 300*fD (by addition)   -->  100.00001192092896
--- Compute 300000*fD (by addition) -->  100000.01192092896
--- fSum (answer should be 100,000) -->  100000.01192092896
```

Representation Errors

Example 2: Experiment with 64-bit floating point numbers.

```
dE = 4.0; dF = 3.0;       # <--- Define two floating point numbers
print("--- dE --> ", dE, " dF --> ", dF );
dG = dE/dF;               # <--- Compute and print dE/dF .....
print("--- Compute dG = (dE/dF)     --> ", dG );
dH = dG-1.0;              # <--- Subtract 1 from dE/dF ...
print("--- Compute dH = (dE/dF - 1) --> ", dH );
print("--- Compute 300*dH           --> ", 300*dH );
```

Output:

```
--- dE -->  4.0, dF -->  3.0
--- Compute dG = (dE/dF)     -->  1.3333333333333333
--- Compute dH = (dE/dF - 1) -->  0.33333333333333326
--- Compute 300*dH           -->  99.99999999999997
--- Compute dSum (300*dD)    -->  99.99999999999966
--- 300000*dH          -->  100000.01192092896
--- dSum (300,000*fH) -->  99999.99999968921
```

Representation Errors

Example 3: Experiments with 0.25 (exact representation in binary)

```
dFraction = 0.25;
print("--- dFraction --> ", dFraction );
dSum = 0.0;
items = range(1,11)            # <--- sum 0.25 ten times ...
for item in enumerate(items):
    dSum = dSum + dFraction;

print("--- Sum 0.25 ten times                    --> ", dSum );
....
print("--- Sum 0.25 one million times            --> ", dSum );
print("--- Sum 0.25 ten million times            --> ", dSum );
print("--- Sum 0.25 one hundred million times --> ", dSum );
```

Output:

```
--- dFraction -->  0.25
--- Sum 0.10 ten times                    -->  2.5
--- Sum 0.25 one million times            -->  250000.0
--- Sum 0.25 ten million times            -->  2500000.0
--- Sum 0.25 one hundred million times -->  25000000.0
```

Representation Errors

Example 4: Experiments with 0.10 (inexact representation)

```
dFraction = 0.1;
print("--- dFraction --> ", dFraction );
dSum = 0.0; # <--- Sum 0.10 ten times ...
items = range(1,11)
for item in enumerate(items):
    dSum = dSum + dFraction;

print("--- Sum 0.10 ten times                    --> ", dSum );
...
print("--- Sum 0.10 one million times            --> ", dSum );
print("--- Sum 0.10 ten million times            --> ", dSum );
print("--- Sum 0.10 one hundred million times --> ", dSum );
```

Output:

```
--- dFraction -->  0.1
--- Sum 0.10 ten times                    -->  0.999999999999999
--- Sum 0.10 one million times            -->  100000.00000133288
--- Sum 0.10 ten million times            -->  999999.9998389754
--- Sum 0.10 one hundred million times -->  9999999.98112945
```

Roundoff Errors

Roundoff Errors

Roundoff errors occur when a floating-point number is rounded to a predetermined level of detail (i.e., not the exact value).

Example 5: Experiments with π.

```
1       # Define approximate value of pi ....
2
3       a = math.pi;
4
5       # Demo floating point formats ...
6
7       print("--- Part 1: Demo floating point formats ...");
8
9       print("---   Maximum detail --> {:.16f}".format(a));
10      print("---   Print 2 decimal places of accuracy --> {:.2f}".format(a));
11      print("---   Print 3 decimal places of accuracy --> {:.3f}".format(a));
12      print("---   Print 4 decimal places of accuracy --> {:.4f}".format(a));
13      print("---   Print 5 decimal places of accuracy --> {:.5f}".format(a));
14
15      # Round floating point numbers ...
16
17      print("--- Part 2: Round floating point formats ...");
18      print("---   Round number (1 decimal places) --> {:f}".format( round(a,1) ));
```

Roundoff Errors

```
20      print("---   Round number (2 decimal places) --> {:f}".format( round(a,2) ));
21      print("---   Round number (3 decimal places) --> {:f}".format( round(a,3) ));
22      print("---   Round number (4 decimal places) --> {:f}".format( round(a,4) ));
23      print("---   Round number (5 decimal places) --> {:f}".format( round(a,5) ));
```

Output:

```
--- Part 1: Demo floating point formats ...
---   Maximum detail --> 3.1415926535897931
---   Print 2 decimal places of accuracy --> 3.14
---   Print 3 decimal places of accuracy --> 3.142
---   Print 4 decimal places of accuracy --> 3.1416
---   Print 5 decimal places of accuracy --> 3.14159

--- Part 2: Round floating point formats ...
---   Round number (1 decimal places) --> 3.100000
---   Round number (2 decimal places) --> 3.140000
---   Round number (3 decimal places) --> 3.142000
---   Round number (4 decimal places) --> 3.141600
---   Round number (5 decimal places) --> 3.141590
```

Approximation Errors

Approximation Errors

Approximation Errors

An approximation error is the discrepancy between an exact value and some approximation to it. The error can be expressed in absolute and relative terms.

Absolute and Relative Errors

Let: (1) x represent the exact answer to a problem, and y a numerical approximation of the exact answer. Two definitions:

$$\text{Absolute error} = |x - y| \,. \tag{2}$$

$$\text{Relative error} = \left[\frac{|x - y|}{|x|} \right] . \tag{3}$$

Approximation Errors

Absolute Error

The absolute error is a measure of how large an error is from a true (or exact) value.

Relative Error

The relative error is a measure of how large an error is relative to the correct (exact) value.

In numerical computations, absolute and relative errors are caused by accumulations of roundoff and/or truncation errors.

Approximation Errors

Significant Digits of Accuracy

We say that:

- A numerical approximation is correct to n significant digits when the first n digits of y agree with the exact answer (x), rounded to n digits.

Key Challenge:

What happens when an exact solution is not known?

Strategy: Let x_n and x_{n+1} be successive estimates of a solution:

- The approximate error is $[x_{n+1} - x_n]$.
- The relative approximate error is $\frac{(x_{n+1}-x_n)}{x_{n+1}} = 1 - \frac{x_n}{x_{n+1}}$.

Preventing Loss of Accuracy

When is Loss of Accuracy a Problem?

Subtractive Cancellation

When nearly equal numbers are subtracted there is a loss of details in the significant digits. This process is called subtractive cancellation.

Rule of Thumb. Take $\log_{10}$ of the ratio largest/smallest intermediate results – this is how many digits could be lost due to subtractive cancellation.

Example. Let smallest: 3.4×10^{-3}, largest: 3.5×10^{6}.

Ratio largest/smallest $\approx 10^{9}$..

Hence, up to 9 digits may have been lost – remaining accuracy is non-warranted (garbage).

When is Loss of Accuracy a Problem?

Practical Computations:

To avoid subtractive cancellation:

- Replace formula by mathematically equivalent formula, having no subtractive cancellation.
- Replace formula by an approximate formula, having no subtractive cancellation.

Demonstrate Loss of Accuracy

Example 1. Consider the expression:

$$f_1(x) = \left[\frac{1 - cos(x)}{x^2}\right] \tag{4}$$

Theoretical Considerations. Recall:

$$cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots \tag{5}$$

Hence, $\lim_{x \to 0} f_1(x) = 0.5$.

Hand Calculation (64-bit arithmetic):

- x $= 10^{-6}$, f_1(x) $=$ 0.500044450, accurate value 0.500000000.
- x $= 10^{-8}$, f_1(x) $=$ 0.00000000, accurate value 0.500000000.

Demonstrate Loss of Accuracy

Reformulate $f_1(x)$:

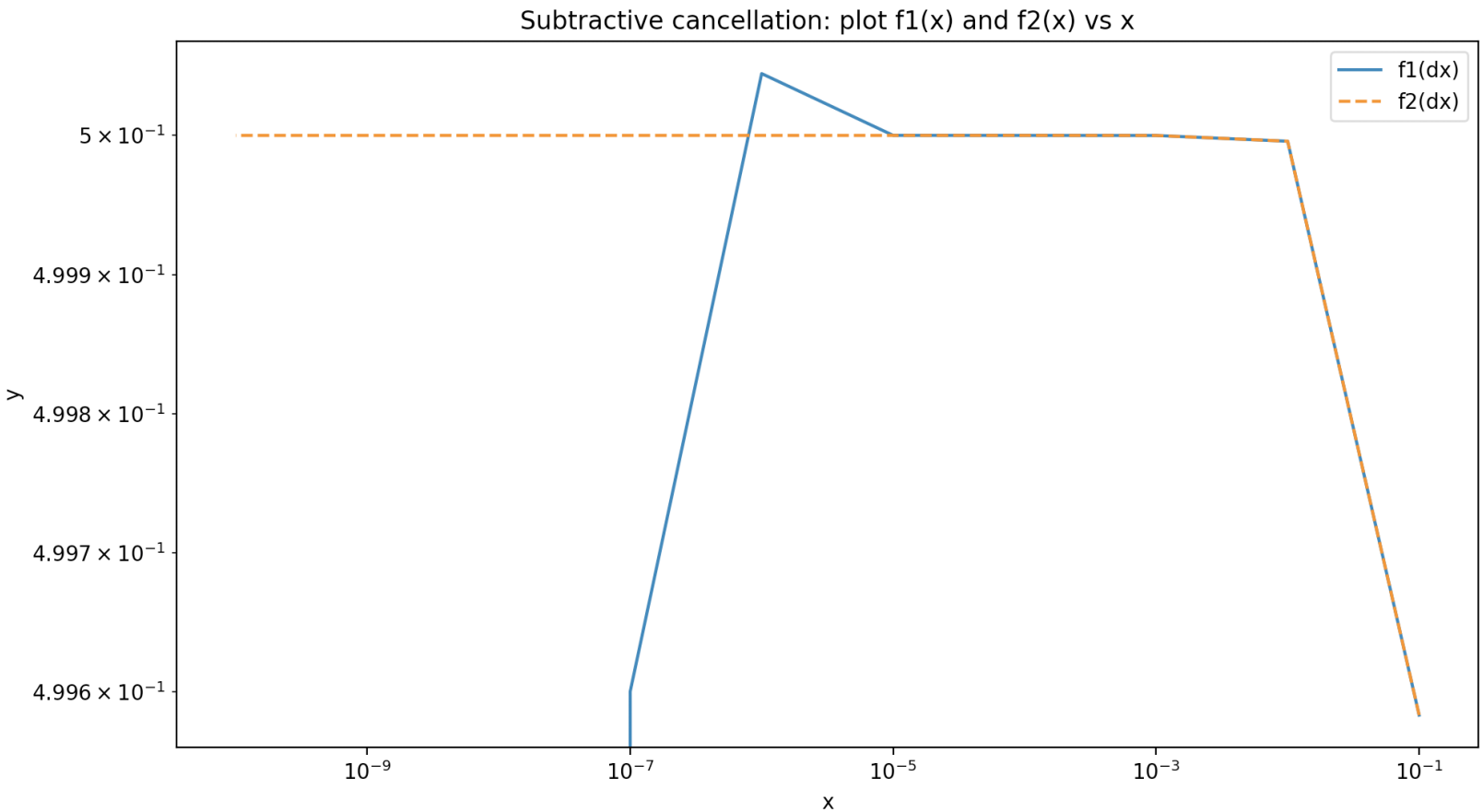
$$f_2(x) = \left[\frac{(1 - cos(x))(1 + cos(x))}{x^2(1 + cos(x))}\right] = \left[\frac{sin^2(x)}{x^2(1 + cos(x))}\right]. \quad (6)$$

Textual Output.

dX	f1(x)	f2(x)
0.1000000000	4.99583472e-01	4.99583472e-01
0.0100000000	4.99995833e-01	4.99995833e-01
0.0010000000	4.99999958e-01	4.99999958e-01
0.0001000000	4.99999997e-01	5.00000000e-01
0.0000100000	5.00000041e-01	5.00000000e-01
0.0000010000	5.00044450e-01	5.00000000e-01
0.0000001000	4.99600361e-01	5.00000000e-01
0.0000000100	0.00000000e+00	5.00000000e-01
0.0000000010	0.00000000e+00	5.00000000e-01
0.0000000001	0.00000000e+00	5.00000000e-01

Demonstrate Loss of Accuracy

Graphical Output.



Prevent Loss of Accuracy

Example 2. Consider the expression:

$$f_3(x) = \left[\frac{cosh(x) - cos(x)}{x^3}\right] \tag{7}$$

Theoretical Considerations. Recall:

$$cosh(x) = 1 + \frac{x^2}{2!} + \frac{x^4}{4!} + \frac{x^6}{6!} + \cdots \tag{8}$$

Hence, $\lim_{x \to 0} f_3(x) = 1/x$.

Hand Calculation (64-bit arithmetic):

- x $= 10^{-6}$, f_3(x) = 1.0000889e+06 (accurate 1.00000e+06).
- x $= 10^{-8}$, f_3(x) = 0.0000000e+00 (accurate 1.00000e+08).

Prevent Loss of Accuracy

Reformulate. Replace $f_3(x)$ with approximate formula:

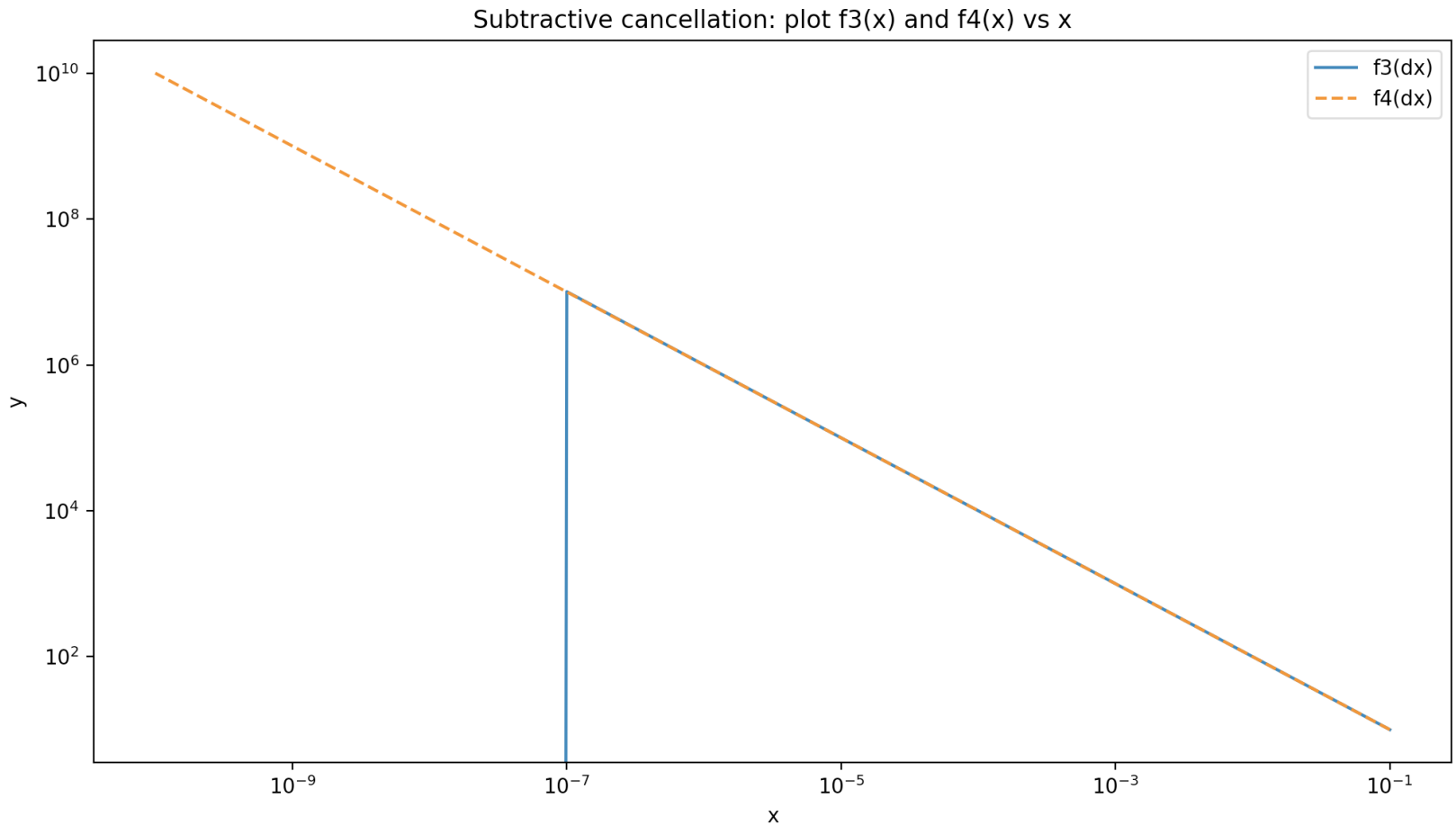
$$f_4(x) = \left[\frac{1}{x}\right]. \tag{9}$$

Textual Output.

dX	f3(x)	f4(x)
0.1000000000	1.00000028e+01	1.00000000e+01
0.0100000000	1.00000000e+02	1.00000000e+02
0.0010000000	1.00000000e+03	1.00000000e+03
0.0001000000	9.99999994e+03	1.00000000e+04
0.0000100000	1.00000008e+05	1.00000000e+05
0.0000010000	1.00008890e+06	1.00000000e+06
0.0000001000	1.01030295e+07	1.00000000e+07
0.0000000100	0.00000000e+00	1.00000000e+08
0.0000000010	0.00000000e+00	1.00000000e+09
0.0000000001	0.00000000e+00	1.00000000e+10

Prevent Loss of Accuracy

Graphical Output.



Python Code Listings

Code 1: Finite Precision Representation of Numbers

```
1  # ======================================================================
2  # TestNumericalPrecision.py: Simple experiments to explore limitations of
3  # finite precision representation of numbers and arithmetic.
4  #
5  # Note. Binary representation of fraction 0.10 has an infinite series.
6  #
7  # Written By: Mark Austin                                  February 2007
8  # ======================================================================
9
10 import math;
11 import numpy;
12
13 # main method ...
14
15 def main():
16     print("--- Enter TestNumericalPrecision.main()               ... ");
17     print("--- ================================================ ... ");
18
19     print("");
20     print("Part 1. Experiment with two 32-bit floating point nos");
21     print("=====================================================");
22
23     #  Define two floating point numbers
24
25     fA = numpy.float32(4);
26     fB = numpy.float32(3)
27
28     # Print variables to default accuracy
```

Code 1: Finite Precision Representation of Numbers

```
29
30      print("--- Print default values of variables");
31      print("--- fA --> ", fA );
32      print("--- fB --> ", fB );
33
34      # Compute and print fA/fB .....
35
36      fC = fA/fB;
37      print("--- Compute fC = (fA/fB) --> ", fC );
38
39      # Now subtract one fA/fB .....
40
41      fD = (fC-1.0);
42      print("--- Compute fD = (fA/fB - 1) --> ", fD );
43
44      # Compute 300*fD (answer should be 100) ...
45
46      print("--- Compute 300*fD (answer should be 100) --> ", 300*fD );
47
48      # Compute sum through addition ....
49
50      fSum  = numpy.float32(0)
51      items = range(1,301)
52      for item in enumerate(items):
53          fSum = fSum + fD;
54
55      print("--- Compute fSum = 300*fD (by addition)   --> ", fSum );
56
57      # ========================================================
58      # Multiply fD by 300,000 (the answer should be 100,000).
```

Code 1: Finite Precision Representation of Numbers

```
59      # ========================================================
60
61      print("--- Compute 300000*fD (by addition) --> ", 300000*fD );
62
63      # Compute sum through addition ....
64
65      fSum  = numpy.float32(0)
66      items = range(1,300001)
67      for item in enumerate(items):
68          fSum = fSum + fD;
69
70      print("--- fSum (answer should be 100,000) --> ", fSum );
71
72      print("");
73      print("Part 2. Experiment with two 64-bit floating point nos");
74      print("========================================================");
75
76      # Define two floating point numbers
77
78      dE = 4.0; dF = 3.0;
79
80      # Print variables to default accuracy
81
82      print("--- Print default values of variables");
83      print("--- dE --> ", dE );
84      print("--- dF --> ", dF );
85
86      # Compute and print dE/dF .....
```

Code 1: Finite Precision Representation of Numbers

```
88      dG = dE/dF;
89      print("--- Compute dG = (dE/dF)     --> ", dG );
90
91      # Now subtract one dE/dF .....
92
93      dH = dG-1.0;
94      print("--- Compute dH = (dE/dF - 1) --> ", dH );
95
96      # ================================================
97      # Multiply dH by 300 (the answer should be 100).
98      # ================================================
99
100     print("--- Compute 300*dH           --> ", 300*dH );
101
102     # Compute sum through addition ....
103
104     dSum  = 0.0
105     items = range(1,301)
106     for item in enumerate(items):
107         dSum = dSum + dH;
108
109     print("--- Compute dSum (300*dD)    --> ", dSum );
110
111     # ========================================================
112     # Multiply dH by 300,000 (the answer should be 100,000).
113     # ========================================================
114
115     print("--- 300000*dH          --> ", 300000*fD );
```

Code 1: Finite Precision Representation of Numbers

```
117         # Compute sum through addition ....
118
119         dSum  = 0.0
120         items = range(1,300001)
121         for item in enumerate(items):
122             dSum = dSum + dH;
123
124         print("--- dSum (300,000*fH) --> ", dSum );
125
126         print("");
127         print("Part 3. Experiment with fraction 0.25");
128         print("=====================================================");
129
130         dFraction = 0.25;
131         print("--- dFraction --> ", dFraction );
132
133         # Sum 0.25 ten times ...
134
135         dSum = 0.0;
136         items = range(1,11)
137         for item in enumerate(items):
138             dSum = dSum + dFraction;
139
140         print("--- Sum 0.10 ten times                    --> ", dSum );
141
142         # Sum 0.25 one million times ...
143
144         dSum = 0.0;
145         items = range(1,1000001)
```

Code 1: Finite Precision Representation of Numbers

```
146         for item in enumerate(items):
147             dSum = dSum + dFraction;
148 
149         print("--- Sum 0.25 one million times          --> ", dSum );
150 
151         # Sum 0.25 ten million times ...
152 
153         dSum = 0.0;
154         items = range(1,10000001)
155         for item in enumerate(items):
156             dSum = dSum + dFraction;
157 
158         print("--- Sum 0.25 ten million times          --> ", dSum );
159 
160         # Sum 0.25 one hundred million times ...
161 
162         dSum = 0.0;
163         items = range(1,100000001)
164         for item in enumerate(items):
165             dSum = dSum + dFraction;
166 
167         print("--- Sum 0.25 one hundred million times --> ", dSum );
168 
169         print("");
170         print("Part 4. Experiment with fraction 0.10");
171         print("=====================================================");
172 
173         dFraction = 0.1;
174         print("--- dFraction --> ", dFraction );
```

Code 1: Finite Precision Representation of Numbers

```
175
176         # Sum 0.10 ten times ...
177
178         dSum = 0.0;
179         items = range(1,11)
180         for item in enumerate(items):
181             dSum = dSum + dFraction;
182
183         print("--- Sum 0.10 ten times                    --> ", dSum );
184
185         # Sum 0.10 one million times ...
186
187         dSum = 0.0;
188         items = range(1,1000001)
189         for item in enumerate(items):
190             dSum = dSum + dFraction;
191
192         print("--- Sum 0.10 one million times            --> ", dSum );
193
194         # Sum 0.10 ten million times ...
195
196         dSum = 0.0;
197         items = range(1,10000001)
198         for item in enumerate(items):
199             dSum = dSum + dFraction;
200
201         print("--- Sum 0.10 ten million times            --> ", dSum );
202
203         # Sum 0.10 one hundred million times ...
```

Code 1: Finite Precision Representation of Numbers

```
204
205         dSum = 0.0;
206         items = range(1,100000001)
207         for item in enumerate(items):
208             dSum = dSum + dFraction;
209
210         print("--- Sum 0.10 one hundred million times --> ", dSum );
211
212         print("--- ================================================ ... ");
213         print("--- Leave TestNumericalPrecision.main()              ... ");
214
215     # call the main method ...
216
217     main()
```

Code 1: Finite Precision Representation of Numbers

Abbreviated Output:

```
Part 1. Experiment with two 32-bit floating point nos
=====================================================
--- Print default values of variables
--- fA -->  4.0, fB -->  3.0
--- Compute fC = (fA/fB) -->  1.3333334
--- Compute fD = (fA/fB - 1) -->  0.3333333730697632
--- Compute 300*fD (answer should be 100) -->  100.00001192092896
--- Compute fSum = 300*fD (by addition)   -->  100.00001192092896
--- Compute 300000*fD (by addition) -->  100000.01192092896
--- fSum (answer should be 100,000) -->  100000.01192092896

Part 2. Experiment with two 64-bit floating point nos
=====================================================
--- Print default values of variables
--- dE -->  4.0, dF -->  3.0
--- Compute dG = (dE/dF)     -->  1.3333333333333333
--- Compute dH = (dE/dF - 1) -->  0.33333333333333326
--- Compute 300*dH           -->  99.99999999999997
--- Compute dSum (300*dD)    -->  99.99999999999966
--- 300000*dH          -->  100000.01192092896
--- dSum (300,000*fH) -->  99999.99999968921
```

Code 1: Truncation/Roundofff Errors

```
Part 3. Experiment with fraction 0.25
=====================================================
--- dFraction -->  0.25
--- Sum 0.10 ten times                    -->  2.5
--- Sum 0.25 one million times            -->  250000.0
--- Sum 0.25 ten million times            -->  2500000.0
--- Sum 0.25 one hundred million times -->  25000000.0

Part 4. Experiment with fraction 0.10
=====================================================
--- dFraction -->  0.1
--- Sum 0.10 ten times                    -->  0.9999999999999999
--- Sum 0.10 one million times            -->  100000.00000133288
--- Sum 0.10 ten million times            -->  999999.9998389754
--- Sum 0.10 one hundred million times -->  9999999.98112945
--- ================================================== ...
--- Leave TestNumericalPrecision.main()                ...
```

Code 2: Subtractive Cancellation

```
1  # ======================================================================
2  # TestSubtractiveCancellation.py: Simple experiments to demonstrate subtractive
3  #                                 cancellation...
4  #
5  # Written By: Mark Austin                                    February 2007
6  # ======================================================================
7
8  import math;
9  import numpy as np
10 from matplotlib import pyplot as plt
11
12 # Define mathematical functions for Part 1....
13
14 def f1 ( dX ):
15     return (1.0 - math.cos( dX ))/( dX*dX );
16
17 def f2 ( dX ):
18     return (math.sin(dX)*math.sin(dX))/(dX*dX*(1.0 + math.cos( dX )));
19
20 # Define mathematical functions for Part 2....
21
22 def f3 ( dX ):
23     return (math.cosh(dX) - math.cos( dX ))/( dX*dX*dX );
24
25 def f4 ( dX ):
26     return (1.0/dX);
27
28 # main method ...
```

Code 2: Subtractive Cancellation

```
29
30  def main():
31      print("--- Enter TestSubtractiveCancellation.main()  ... ");
32      print("--- ========================================= ... ");
33
34      # Generate list of x values ....
35
36      print("--- Part 1. Generate and print list of x values ... ");
37
38      xcoord = []
39      xcoord.append ( 0.1 )
40      xcoord.append ( 0.01 )
41      xcoord.append ( 0.001 )
42      xcoord.append ( 0.0001 )
43      xcoord.append ( 0.00001 )
44      xcoord.append ( 0.000001 )
45      xcoord.append ( 0.0000001 )
46      xcoord.append ( 0.00000001 )
47      xcoord.append ( 0.000000001 )
48      xcoord.append ( 0.0000000001 )
49      print(xcoord)
50
51      print("");
52      print("--- Part 2. Experiment with:  ");
53      print("   f1(x) = (1-cos(x))/(x*x) ");
54      print("   f2(x) = (sin(x)^2)/(x*x)(1+cos(x)) ");
55      print("================================================");
56      print("              dX             f1(x)             f2(x)");
```

Code 2: Subtractive Cancellation

```
57        print("================================================");
58
59        ycoord1 = []
60        ycoord2 = []
61        for dX in xcoord:
62           ycoord1.append( f1(dX) )
63           ycoord2.append( f2(dX) )
64           print("{:16.10f} {:16.8e} {:16.8e}".format( dX, f1(dX), f2(dX)));
65
66        print("================================================");
67
68        print("");
69        print("--- Part 3. Experiment with:              ");
70        print("   f3(x) = (cosh(x)-cos(x))/(x*x*x) ");
71        print("   f4(x) = 1/x                      ");
72        print("================================================");
73        print("               dX             f3(x)             f4(x)");
74        print("================================================");
75
76        # Compute and print function values....
77
78        ycoord3 = []
79        ycoord4 = []
80        for dX in xcoord:
81           ycoord3.append( f3(dX) )
82           ycoord4.append( f4(dX) )
```

Code 2: Subtractive Cancellation

```
83          print("{:16.10f} {:16.8e} {:16.8e}".format( dX, f3(dX), f4(dX)));
84
85      print("");
86      print("--- Part 4. Plot numerical experiment f1(x) and f2(x) vs x ... ");
87
88      plt.plot(xcoord, ycoord1, label="f1(dx)", linestyle="-")
89      plt.plot(xcoord, ycoord2, label="f2(dx)", linestyle="--")
90      plt.title("Subtractive cancellation: plot f1(x) and f2(x) vs x")
91      plt.xscale("log")
92      plt.yscale("log")
93      plt.xlabel('x')
94      plt.ylabel('y')
95      plt.legend()
96      plt.show()
97
98      print("");
99      print("--- Part 5. Plot numerical experiment f3(x) and f4(x) vs x ... ");
100
101     ... source code removed ...
102     ... details are almost identical to Part 4 ...
103
104     print("--- ============================================== ... ");
105     print("--- Leave TestSubtractiveCancellation.main()       ... ");
106
107 # call the main method ...
108
109 main()
```