Numerical Precision and Truncation & Round off Errors

Topics covered

Taylor’s expansion
truncation errors
round-off errors
underflow and overflow
precision of data types C++

Introduction

It is important to understand that, in general, numerical methods are not exact; neither are the machines (computers) that perform the numerical calculations for us. In this lecture, we will look at the nature of truncation errors and round-off errors. An understanding of these sources of errors in numerical methods is as important as an understanding of the methods themselves.

Numerical Methods

We apply numerical techniques to solve numerical problems when analytical solutions are difficult or inconvenient. A simple example is the computation of the first derivative of a function f(x). Calculus gives us an analytical method for forming an expression for the derivative, however, such analysis for some functions may be difficult, impossible, or inconvenient. A simple numerical solution uses the Forward Difference Approximation (FDA) that approximates the derivative by taking the gradient of the function f(x) in the region x to x+h:

\[ \text{FDA} = \frac{f(x+h) - f(x)}{h} \]

where h is small but not zero.

For example if \( f(x) = 2x^2 + 4x + 6 \) and we wish to determine the first derivative evaluated at \( x=3 \), the FDA (using \( h=0.01 \)) gives:

\[ \frac{ (2^3.01^2 + 4^3.01 + 6) - (2^3.00^2 + 4^3.00 + 6) }{0.01} = 16.02. \]

Of course this is only an approximation (the true value, by calculus, is 16).

```
gnuplot> plot [0:4] 2*x**2+4*x+6
```

![Plot of f(x) = 2x^2 + 4x + 6 from x=0 to x=4]

1
Truncation Errors [kesme hatası]

The error in the above approximation can be written as

\[ \text{FDA} - f'(x) = 16.02 - 16 = 0.02. \]

This is called a truncation error as it is due to the truncation of higher orders in the exact expression for the first derivative. We can see the form of the truncation error in the FDA by considering Taylor's expansion:

\[ f(x+h) = f(x) + h.f'(x)/1! + h^2.f''(x)/2! + h^3.f'''(x)/3! + \ldots \]

Rearrange for the FDA:

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

\[ \text{FDA} = f'(x) + (h/2).f''(x) + O(h^2) \]

We see that the FDA gives the first derivative plus some extra terms in the series. The error in the approximation FDA - f'(x) is therefore \((h/2).f''(x) + O(h^2)\). This can be checked numerically with the above example: \((h/2).f''(x) = (0.01/2) * (4) = 0.02\) (as found above).

The truncation error in the FDA is proportional to \(h\), the FDA is therefore called a first-order approximation. Higher-order methods have truncation errors that are proportional to higher powers of \(h\) and therefore yield smaller truncation errors (when \(h\) is less than one). We will investigate the round-off error in the above calculation at the end of the next section.

Computer Precision (Round-off Errors) [yuvarlama hatası]

Numerical methods are implemented in computer programs where the numerical calculations can be performed quickly and conveniently. However, numbers are stored in computer memory with a limited precision; the loss of precision of a value is called a round-off error. Round-off errors can occur when a value is initially assigned, and can be compounded when values are combined in arithmetic operations. Iteration is common in computational methods and so it is important to minimise compound roundoff. As round-off errors can be a significant source of error in a numerical method (in addition to the truncation error) we will look more closely at the nature of the round-off error and how it can be reduced. A binary representation is used to store numbers in computer memory.

For example the binary number 11.011 represents exactly the decimal number 3.375:

\[
11.011 = 1*2 + 1*1 + 0*1/2 + 1*1/4 + 1*1/8 = 3.375
\]
Similarly the decimal value 0.3125 can be expanded to \(0.25 + 0.0625 = 1/4 + 1/16\) that can be stored exactly in binary as 0.0101. However, given a limited number of binary digits, it is possible that even a rational decimal number might not be stored precisely in binary. For example there is no precise representation for 0.3; the nearest representation with 8 bits is 0.01001101 that gives 0.30078125. The precision increases as more binary digits are used, but there is always a round-off error. In general, the only real numbers that can be represented exactly in the computer’s memory are those that can be written in the form \(m/2^k\) where \(m\) and \(k\) are integers; however, again there is a limit to the set of numbers that are included in this group due to the limited number of binary digits used to store the value.

**Floating-Point Representation**

Computers store REAL numbers (as opposed to INTEGER numbers) in the floating point representation:

\[
\text{value} = m \cdot b^e
\]

where \(m\) is the mantissa, \(b\) is the base (\(= 2\) in computers) and \(e\) is the exponent. In C++, float number is stored in 32 binary bits (4 bytes). To allow for a large exponent range the binary bits available for storage are shared between the mantissa and the exponent of the number. For example the number 413.26 is represented by a mantissa part and an exponent part as 0.41326\times10^3. The division of the 32 binary bits are as follows: 8 bits are used for to store the exponent, 1 bit for the sign of the exponent, and 23 bits for the mantissa. The precision of the storage of real data is therefore limited by the 23 bits used to store the mantissa.

In C++ the number of binary bits used to store type float numbers can be increased from the default 32 type double numbers can be increased from the default 64

A limitation is also placed on the range of values that be be stored, this is illustrated for single-precision type double data below:

- **very large -ve**
  - \(-10^{308}\)
  - **overflow!**

- **very small**
  - \(-10^{-324}\)
  - \(10^{-324}\)
  - **Underflow!**

- **very large +ve**
  - \(10^{308}\)
  - **Overflow!**

An **overflow** occurs when a number is too large to be stored. An **underflow** occurs when a number is too small to be stored.

The number zero is an exception –it can be stored exactly. If you exceed the limits of precision or range then your program will give wrong results! C++ provides modifiers to these data types and provides more data types; see the table given below:
The expression \( \frac{(a+b)^2 - 2ab - b^2}{a^2} \) reduces to \( a^2/a^2 = 1 \).

But computed in a machine with limited precision can give unexpected results:

```cpp
#include <iostream>
using namespace std;

int main()
{
    double a=0.00001, b=88888, c;
    c = ( (a+b)*(a+b) - 2*a*b - b*b ) / (a*a);
    std::cout << c << std::endl;
}
```

The output is 4.65661. This is an extreme example of a calculation that is sensitive to round-off.

Note that long double gives the correct result 1.000000000000017.

**EXAMPLE**

What is the output of the following programs?

```cpp
#include <iostream>
using namespace std;

int main()
{
    float hbar = 1.07e-34;
    float me = 9.1e-31;
    cout << 0.5f * hbar*hbar/me;
}
```

**Outputs**

0

```cpp
#include <iostream>
using namespace std;

int main()
{
    double hbar = 1.07e-34;
    double me = 9.1e-31;
    cout << 0.5 * hbar*hbar/me;
}
```

**Output**

6.29066e-39

---

**TABLE 1.1:** Fundamental data types and their size and ranges in the memory. The numbers are evaluated for a 32-bit system.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
<th>Size (byte)</th>
<th>Lower Limit</th>
<th>Upper Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>Character or small integer</td>
<td>1</td>
<td>-128</td>
<td>127</td>
</tr>
<tr>
<td>unsigned char</td>
<td></td>
<td>1</td>
<td>0</td>
<td>255</td>
</tr>
<tr>
<td>short int</td>
<td>Short integer</td>
<td>2</td>
<td>-32,767</td>
<td>32,767</td>
</tr>
<tr>
<td>unsigned short int</td>
<td></td>
<td>2</td>
<td>0</td>
<td>65,535</td>
</tr>
<tr>
<td>int</td>
<td>Integer</td>
<td>4</td>
<td>-2,147,483,648</td>
<td>2,147,483,647</td>
</tr>
<tr>
<td>unsigned int</td>
<td></td>
<td>4</td>
<td>0</td>
<td>4,294,967,295</td>
</tr>
<tr>
<td>unsigned long int</td>
<td></td>
<td>8</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>float</td>
<td>Single precision floating point number (7 digits)</td>
<td>4</td>
<td>-1.0e+30</td>
<td>+1.0e+30</td>
</tr>
<tr>
<td>double</td>
<td>Double precision floating point number (15 digits)</td>
<td>8</td>
<td>-5.0e+30</td>
<td>+5.0e+30</td>
</tr>
<tr>
<td>long double</td>
<td>Quad precision floating point number (34 digits)[*]</td>
<td>16</td>
<td>-1.0e+4931</td>
<td>+1.0e+4931</td>
</tr>
</tbody>
</table>

[*] only on 64 bit platforms.
Exercises

1. Series expansion of the sine function is given by

\[ \sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots \]

By using only the first term of the expansion, write a computer program to evaluate the truncation errors for \( x = 0.01, 0.1 \) and 1.0.

2. What do you expect to be the output of the following program section:
Run the program to see if you are right, explain your findings.

```c
int main(){
    double a=0.0;
    while(1){
        a = a + 0.6;
        cout << a << endl;
        if ( a == 1.8 ) break;
    }
}
```