Notes for Numerical Analysis
Math 4445
by
S. Adjerid
Virginia Polytechnic Institute
and State University

(A Rough Draft)
Contents

1 Error Analysis
   1.1 Representation of numbers and round-off errors . . . . . . . 5
      1.1.1 Representation of integers . . . . . . . . . . . . . . . . 5
      1.1.2 Floating-point representation . . . . . . . . . . . . . 7
      1.1.3 Chopping versus rounding . . . . . . . . . . . . . . . 11
      1.1.4 Negative effects of round-off errors: . . . . . . . . . . 12
      1.1.5 Other types of errors . . . . . . . . . . . . . . . . . . 14
   1.2 Error propagation and stability . . . . . . . . . . . . . . . 16
Chapter 1

Error Analysis

1.1 Representation of numbers and round-off errors

In this section we discuss how numbers are represented on computers and how round-off errors are introduced in the calculations.

1.1.1 Representation of integers

Computers represent integer numbers using a finite number of digits in the binary system (base 2). We start with a number in the decimal system (base 10)

\[2583 = 2 \times 10^3 + 5 \times 10^2 + 8 \times 10^1 + 3 \times 10^0\]

In other words 2 is the number of thousands, 5 is the number of hundreds, 8 number of tens and 3 is the number of ones.

In the binary system (base 2) numbers are represented by a string of 1 or 0 as illustrated in the following example

\[(110110)_2 = 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0\]

An algorithm to convert a integer from the decimal system to the binary system reads as follows

For \(A = 31\), the algorithm takes the following steps
function B=integerconvert10to2(D)
% convert a integer number D from decimal
% to binary representation
% input: D has to be between -1 < B < 1
% output: B string of 0 or 1
s = sign(D);
if s == 1
    B(1)= 0;
else
    B(1) =1;
end
%
D = abs(D);
n=1;
ndigits=32;
while (n < ndigits )
    B(ndigits+1-n) = mod(D,2);
    D = fix(D/2);
    n=n+1;
end
31 = 2 * 15 + 1
15 = 2 * 15 + 1
7 = 2 * 3 + 1
3 = 2 * 1 + 1
1 = 2 * 0 + 1

Thus we can write

\[(31)_{10} = (11111)_2\]

Since computers use 32-bit-long words to represent integers, the number

\[A = (-1)^s d_{30}2^{30} + d_{29}2^{29} + \cdots + d_12^1 + d_0, \quad s = 0, 1.\]

is represented by the following string

\[
\begin{array}{cccccc}
\text{s} & d_{30} & d_{29} & \cdots & d_1 & d_0 \\
\end{array}
\]

The first bit contains the sign information \(s = 0\) for + and \(s = 1\) for -, the
31 next bits contain the digits of the number.

Given the finite number of digits used to represent integers we note that a
number greater than \(2^{30} + 2^{29} + \cdots + 2 + 1\) will cause overflow.

### 1.1.2 Floating-point representation

Numbers with decimal part are written in a normalized floating-point form
where only significant digits are stored. For instance in the decimal system
00457.98 is stored as 0.4579810\(^5\)
0.00000389 is stored as 0.38910\(^{-5}\)

The general normalized floating-point representation of numbers in the dec-
imal system have the form

\[
\pm(0.d_1d_2d_3\cdots d_n\cdots) \times 10^m =
\pm(d_1 \times 10^{-1} + d_2 \times 10^{-2} + d_3 \times 10^{-3} \cdots + d_n \times 10^{-n} \cdots) \times 10^m
\]

In the binary system the general form is

\[A = \pm(0.b_1b_2b_3\cdots b_n\cdots) \times 2^m =
\pm(b_1 \times 2^{-1} + b_2 \times 2^{-2} + b_32^{-3} + \cdots + b_n \times 2^{-n} \cdots) \times 2^m.\]

Let \(\text{int}(A)\) denote the integer part of \(A\) and \(\text{fr}(A)\) denote the decimal part of
\(A\).

*Remarks:*
Some numbers are represented by an infinite sequence of digits such as $1/3 = (0.333333\cdots)_{10}$

Some numbers are represented by a finite number of digits in one system and require an infinite sequence of digits in another system as illustrated in the example

$$(0.7)_{10} = (0.10110011001100110\cdots)_2$$

Let us give more examples of normalized binary numbers

$$(1.00111)_2 = (0.100111)_2 \times 2$$

$$(1011.011)_2 = (0.1011011)_2 \times 2^4$$

$$(0.0000110)_2 = (0.110)_2 \times 2^{-4}$$

**A Conversion Algorithm:**

Next we describe an algorithm that converts the fractional part of a floating-point from the decimal system to the binary system. Let $A$ be a number between 0 and 1 such as

$$A = (0.d_1 d_2 d_3 \cdots d_n)_{10}$$

which can be written as

$$A = \frac{b_1}{2} + \frac{b_2}{2^2} + \frac{b_3}{2^3} + \cdots + \frac{b_n}{2^n} + \cdots$$

$$2A = b_1 + \frac{b_2}{2} + \frac{b_3}{2^2} + \cdots + \frac{b_n}{2^{n-1}} + \cdots.$$

Therefore, we can write $b_1 = \text{floor}(2A)$, $b_2 = \text{floor}(2(2A - b_1))$. Consult the matlab script below.

**A Conversion Algorithm in Matlab**

```matlab
function B=floatconvert10to2(D)
% convert a floating-point number D from decimal
% to binary representation
```
1.1. REPRESENTATION OF NUMBERS AND ROUND-OFF ERRORS

% input: D has to be between -1 < D < 1
% output: B string of 0 or 1
s = sign(D);
if s == 1
    b(1) = 0;
else
    b(1) = 1;
end
%
D = abs(D);
n=1;
ndigits = 24;
while (n <= ndigits )
    B(8+n) = floor(2*D);
    D = 2*D - B(8+n);
    n=n+1;
end

The general form of a floating-point representation in a system of base $a$ with $k$ significant digits takes the form

$$A = \pm 0.d_1 d_2 d_3 \cdots d_k a^t$$

Where $0.d_1 d_2 d_3 \cdots d_k$ is the mantissa and $a^t$ is the characteristic.

**Standard IEEE representation:**
For instance, in the standard IEEE floating-point representation that most modern computers use, the string of binary digits

**Single precision:**

$$b_1 b_2 b_3 \cdots b_{32}$$

is interpreted as

$$(-1)^{b_1} 2^{(b_2 b_3 \cdots b_9)} 2^{-127} \times 1.b_{10} b_{11} \cdots b_{32}$$

The characteristic is bounded as

$$c = (b_2 b_3 \cdots b_9)_2, \text{ where } 0 < c < (11111111)_2 = 255.$$
CHAPTER 1. ERROR ANALYSIS

- \( c = 0 \) reserved for zero
- \( c = 255 \) reserved for \( \pm \infty \)
- \( c = 1 \) corresponds to the smallest computer number \( \approx 2^{-126} \approx 1.2 \times 10^{-38} \)
- \( c = 254 \) corresponds to the largest computer number \( (2 - 2^{-23}) \times 2^{127} \approx 3.4 \times 10^{38} \) where we used \( \sum_{n=0}^{23} 2^{-n} \).
- NaN where \( c = 255 \) and \( f \neq 0 \) corresponds to \( 0/0 = \infty - \infty \) and \( x + NaN \).
- Very small negative numbers are represented by \( -0 \)
- Most operations yielding \( +0 \)
- \( x/\infty \) yields \( +0 \)
- \( x + \infty, \infty/x \) yields \( +\infty \)

Double or extended precision
The following string of 64 digits

\[ b_1 | b_2 b_3 \cdots b_{12} | b_{13} \cdots b_{64} \]

is interpreted as

\[ (-1)^{b_1} \times 2^{(b_2 b_3 \cdots b_{12})} \times 2^{-1023} \times 1.0 \times b_{13} \cdots b_{64} \]

where

- \( c = (b_2 b_3 \cdots b_{12})_2 \), where \( 0 < c < (1111111111)_2 = 1047 \).
- \( c = 1 \) corresponds to the smallest number which \( \approx 0.22210^{-307} \)
- \( c = 2046 \) corresponds the largest number \( \approx 1.7910^{309} \).
1.1. REPRESENTATION OF NUMBERS AND ROUND-OFF ERRORS

1.1.3 Chopping versus rounding

There are two common ways to approximate floating-point numbers with a finite number of digits.

Chopping: The number is chopped at a given location to keep a fixed number of significant digits as

\[ \frac{2}{3} = (0.666666666666...)_{10} \approx (0.6666666..6)_{10}. \]

In general, chopping of the number

\[ A = \pm (0.d_1d_2\ldots d_n\ldots) \times 10^t \]

leads to

\[ A \approx \tilde{A} = \pm (0.d_1d_2\ldots d_k) \times 10^t. \]

The chopping relative error can be estimated as

\[ \frac{|A - \tilde{A}|}{|A|} = \frac{0.d_{k+1}\ldots 10^{t-k} \times 10^t}{0.d_1d_2\ldots d_k10^t} < \frac{10^{t-k}}{10^{t-1}} = 10^{1-k} \]

Rounding:

A number is rounded according to the following rule

\[ A \approx \tilde{A} = \begin{cases} (0.d_1d_2\ldots d_k) \times 10^t + 0.000\ldots 01 \times 10^t, & \text{if } d_{k+1} \geq 5 \\ (0.d_1d_2\ldots d_k) \times 10^t, & \text{Otherwise}. \end{cases} \]

The absolute rounding error can be estimated as

\[ |A - \tilde{A}| < 0.510^{t-k}. \]

Since \( |A| > 10^{t-1} \), the relative rounding error satisfies

\[ \frac{|A - \tilde{A}|}{|A|} < \frac{0.510^{t-k}}{10^{t-1}} = 0.510^{1-k}. \]

We note that rounding errors are smaller than chopping errors.

Machine precision: Let

\[ x = 0.b_1b_2\ldots b_kb_{k+1}\ldots 2^t \]
Chopping with \( k \) significant digits we have

\[
chop(x) = 0.b_1b_2\cdots b_k2^t
\]

the chopping error can be written as

\[
\frac{|x - chop(x)|}{|x|} = \frac{0.b_{k+1}\cdots 2^{-k}2^t}{0.b_1b_2\cdots 2^t} < \frac{2^{-k}}{1/2}.
\]

Thus,

\[
\frac{|x - chop(x)|}{|x|} < 2 \cdot 2^{-k}
\]

Rounding yields

\[
round(x) = \begin{cases} 
0.b_1b_2\cdots b_k2^t, & \text{if } b_{k+1} = 0 \\
(0.b_1b_2\cdots b_k + 0.00\cdots 1)2^t, & \text{otherwise}
\end{cases}
\]

The rounding error can be written as

\[
\frac{|x - round(x)|}{|x|} < \frac{0.1\cdots 2^{-k}2^t}{0.b_1b_2\cdots 2^t} < \frac{2^{-k-1}2^t}{2^t1/2}
\]

Thus, we can write

\[
\frac{|x - round(x)|}{|x|} < 2^{-k}
\]

where \( k \) is the number of significant digits.

Thus, the machine precision is \( Eps = 2^{-k} \).

Now any number can be expressed as

\[
round(x) = x(1 + \epsilon), \quad |\epsilon| \leq Eps.
\]

Single precision: \( k = 24, Eps = 2^{-24} \approx 10^{-8} \)

Double precision: \( k = 53 \) and \( Eps = 2^{-53} \approx 10^{-16} \).

1.1.4 Negative effects of round-off errors:

Now, we will consider an example to illustrate the effect of rounding errors.

We consider \( a = 56.02, b = 0.002, c = 0.004 \), and use \( k = 4 \) significant digits in the decimal system
Let us compute \((a + b) + c\)
\[a + b = 56.02 + 0.002 = 56.022,\] after rounding we obtain \(a + b = 56.02\)
\[(a + b) + c = 56.02 + 0.004 = 56.024,\] after rounding we obtain 56.02

Now, let us change the order of operation as \((a + (b + c))\)
\[(b + c) = 0.006,\] rounding yields 0.006
\[a + (b + c) = 56.026,\] rounding yields 56.03

We note that we obtain different results by changing the order of operations.

Exercise: Use finite precision arithmetic with \(k = 4\) and the correct order of operations to find the correct value of the following sum

\[1 + \sum_{k=0}^{10^8} 0.00001\]

Loss of significant digits: cancellation

Another problem with using finite precision arithmetic is the loss of significant digits. This mainly occurs during subtraction of nearly equal quantities.

Example: Let us use \(k = 14\) significant digits in the decimal system (double precision) to compute \(\cos(x) - \cos(2x)\) for \(x = 0.0001\).

\[
\begin{align*}
\cos(x) & = 0.99999999500000 \\
\cos(2x) & = 0.99999998000000 \\
\cos(x) - \cos(2x) & = 0.000000014999999908379 \\
\cos(x) - \cos(2x) & = 0.149999999088379 \times 10^{-7}
\end{align*}
\]

We observe that there is a lost of seven significant digits.

Avoiding loss of significance:

Let us illustrate this on the expression

\[
\sqrt{1 + x^2} - \sqrt{1 - x^2}
\]

If we use it in this form we will experience a loss of significant digits for values of \(|x| \ll 1\). We can avoid this loss of significance by rewriting the expression it as
to obtain
\[
\frac{x^4 + x^2}{(\sqrt{1 + x^4 + \sqrt{1 - x^2}})}.
\]

1.1.5 Other types of errors

The numerical error is obtained as a result of replacing a continuous mathematical problem with a simpler approximation as illustrated in the following example where we approximate the derivative of a function \( f(x) \) at \( x_0 \) as

\[
f'(x_0) \approx \tilde{f}'(x_0) = \frac{f(x_0 + h) - f(x_0)}{h}, \quad 0 < h << 1.
\]

The numerical error is given by

\[
h \frac{1}{2} f''(\xi), \quad x_0 < \xi < x_0 + h.
\]

In addition to the numerical error there are round-off errors due to computer finite precision arithmetic. Using double precision on Matlab we obtain

\[
\frac{\tilde{f}(x_0 + h) - \tilde{f}(x_0)}{h} = \frac{(f(x_0 + h) + \delta_1) - (f(x_0) + \delta_2)}{h}
\]

Thus the total error is

\[
E = f'(x_0) - \frac{\tilde{f}(x_0 + h) - \tilde{f}(x_0)}{h} = h \frac{1}{2} f''(\xi) + \frac{\delta_1 - \delta_2}{h}.
\]

If we assume that \( \delta_i < \delta \) \( i = 1, 2 \) and \( M_2 = \max_{[x_0, x_0 + h]} |f''(\xi)| \), then we can bound the error as

\[
|E| < h \frac{1}{2} M_2 + \frac{2\delta}{h}.
\]

We remark that for \( h << \delta \) the numerical error dominates and for \( h \approx \delta \) the round-off error dominates.

Example: Let \( f(x) = \sin(x), \quad x_0 = 0.3 \)
1.1. REPRESENTATION OF NUMBERS AND ROUND-OFF ERRORS

\[
\frac{(f(0.3+h)-f(0.3))/h}{h} \quad \text{Error}
\]

<table>
<thead>
<tr>
<th>h</th>
<th>( f(0.3+h)-f(0.3) )</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/2</td>
<td>0.84367176847637</td>
<td>0.11166472064924</td>
</tr>
<tr>
<td>1/4</td>
<td>0.90866808907728</td>
<td>0.0466640004833</td>
</tr>
<tr>
<td>1/8</td>
<td>0.93440460065668</td>
<td>0.02093188846892</td>
</tr>
<tr>
<td>1/16</td>
<td>0.94548264606895</td>
<td>0.00985384305666</td>
</tr>
<tr>
<td>(1/2^-5)</td>
<td>0.95056387828479</td>
<td>0.00477261084081</td>
</tr>
<tr>
<td></td>
<td>0.95298891221562</td>
<td>0.00234757690999</td>
</tr>
<tr>
<td></td>
<td>0.95417240103413</td>
<td>0.00116408809148</td>
</tr>
<tr>
<td></td>
<td>0.95475687241144</td>
<td>0.00057961671417</td>
</tr>
<tr>
<td></td>
<td>0.95504728787907</td>
<td>0.00028920124654</td>
</tr>
<tr>
<td></td>
<td>0.9551924031456</td>
<td>0.0001444881106</td>
</tr>
<tr>
<td></td>
<td>0.95526430267751</td>
<td>0.00007218644810</td>
</tr>
<tr>
<td></td>
<td>0.95530040539143</td>
<td>0.00003608373418</td>
</tr>
<tr>
<td></td>
<td>0.95531844963125</td>
<td>0.00001803949436</td>
</tr>
<tr>
<td></td>
<td>0.95532746997196</td>
<td>0.00000901915364</td>
</tr>
<tr>
<td></td>
<td>0.95533197969780</td>
<td>0.00000450942780</td>
</tr>
<tr>
<td></td>
<td>0.95533423444795</td>
<td>0.00000225467766</td>
</tr>
<tr>
<td></td>
<td>0.95533536179573</td>
<td>0.00000112732987</td>
</tr>
<tr>
<td></td>
<td>0.95533592546417</td>
<td>0.00000056366144</td>
</tr>
<tr>
<td></td>
<td>0.95533620729111</td>
<td>0.00000028183449</td>
</tr>
<tr>
<td></td>
<td>0.9553363821186</td>
<td>0.00000014091375</td>
</tr>
<tr>
<td></td>
<td>0.95533641870134</td>
<td>0.00000007042427</td>
</tr>
<tr>
<td></td>
<td>0.95533645385876</td>
<td>0.00000003526684</td>
</tr>
<tr>
<td></td>
<td>0.95533647155389</td>
<td>0.00000001757171</td>
</tr>
<tr>
<td></td>
<td>0.95533648040146</td>
<td>0.00000000872415</td>
</tr>
<tr>
<td></td>
<td>0.95533648505807</td>
<td>0.00000000406754</td>
</tr>
<tr>
<td></td>
<td>0.95533648878336</td>
<td>0.0000000034225</td>
</tr>
<tr>
<td></td>
<td>0.95533648878336</td>
<td>0.0000000034225</td>
</tr>
<tr>
<td></td>
<td>0.95533649623394</td>
<td>0.00000000710833</td>
</tr>
<tr>
<td></td>
<td>0.9553365113510</td>
<td>0.00000000220095</td>
</tr>
<tr>
<td></td>
<td>0.9553365113510</td>
<td>0.00000000220095</td>
</tr>
<tr>
<td></td>
<td>0.95533657073975</td>
<td>0.00000008161414</td>
</tr>
<tr>
<td></td>
<td>0.95533657073975</td>
<td>0.00000008161414</td>
</tr>
<tr>
<td></td>
<td>0.95533657073975</td>
<td>0.00000008161414</td>
</tr>
</tbody>
</table>
0.95533657073975 0.000000008161414
0.95533752441406 0.000000103528846
0.95533752441406 0.000000103528846
0.95533752441406 0.000000103528846
0.95533752441406 0.000000103528846
0.95535278320312 0.00001629407752
0.95538330078125 0.00004681165564
0.95532265625000 0.00001422350061
0.95532265625000 0.00001422350061
0.95556640625000 0.00022991712439
0.95605468750000 0.00071819837439
0.95507812500000 0.0025836412561
0.95703125000000 0.00169476087439
0.96093750000000 0.00560101087439
0.95312500000000 0.00221148912561
0.96875000000000 0.01341351087439
1.00000000000000 0.04466351087439
1.00000000000000 0.04466351087439

1.2 Error propagation and stability

Now any number \( x \) can be rounded as

\[
\text{round}(x) = x(1 + \epsilon), \quad |\epsilon| \leq Eps.
\]

Machine arithmetic and elementary operations:

\[
\text{round}(x \ast y) = (x \ast y)(1 + \epsilon),
\]

Numerical stability:

\textit{definition}: An algorithm to compute \( f(x) \) is \textit{numerically stable} if and only if

\[
\text{round}(f(x)) = f(x)(1 + \delta),
\]

where \(|\delta| < CEps, C > 0.\)

Examples:
1.2. ERROR PROPAGATION AND STABILITY

**Multiplication:**

\[ x(1 + \epsilon_x)y(1 + \epsilon_y) = xy(1 + \epsilon_x + \epsilon_y + \epsilon_x \epsilon_y) \approx xy(1 + \epsilon_x + \epsilon_y) \]

The relative error is approximated by \( |\epsilon_{xy}| = |\epsilon_x + \epsilon_y| < 2Eps \). Thus multiplication is stable.

**Division:**

\[ \frac{x(1 + \epsilon_x)}{y(1 + \epsilon_y)} = \frac{x}{y}(1 + \epsilon_x)(1 - \epsilon_y + \epsilon_y^2 \cdots) \approx \frac{x}{y}(1 + \epsilon_x - \epsilon_y) \]

Thus, the relative error is approximated by

\[ \epsilon_{x/y} = \epsilon_x - \epsilon_y \]

where \( |\epsilon_{x/y}| < 2Eps \). Thus, division is stable.

**Addition and subtraction:**

\[ x(1 + \epsilon_x) + y(1 + \epsilon_y) = x + y + x\epsilon_x + y\epsilon_y = (x + y)(1 + \frac{x\epsilon_x + y\epsilon_y}{x + y}) \]

assuming \((x + y) \neq 0\) we have

\[ \epsilon_{x+y} = \frac{x\epsilon_x + y\epsilon_y}{x + y} \]

If \(x\) and \(y\) have the same sign we have

\[ |\epsilon_{x+y}| = |\epsilon_x| + |\epsilon_y| < 2Eps \]

The operation is stable.

If \(x\) and \(y\) are of opposite signs and \(x + y\) is arbitrarily small, \(x\) and \(y\) are almost equal in absolute value, then error can be magnified. This is called cancellation error and should be avoided whenever possible.

**Example:**
Consider the problem which consists of computing the sum $s = a + b + c$

**Algorithm 1:** $u = a + b$
$s = u + c$

**Algorithm 2:** $u = b + c$
$s = a + u$

**Algorithm 3:** $u = a + c$
$s = u + b$

For *Algorithm 1*, we have:

$$
\text{round}(s) = \text{round}(a + b + c) = ((a + b)(1 + \epsilon_{a+b}) + c)(1 + \epsilon_{(a+b)+c})$

$$
= (a + b + c)[1 + \frac{a + b}{a + b + c}\epsilon_{a+b}(1 + \epsilon_{(a+b)+c}) + \epsilon_{(a+b)+c}].
$$

The relative error:

$$
\tilde{\epsilon}_{(a+b)+c} = \frac{a + b}{a + b + c}\epsilon_{a+b}(1 + \epsilon_{(a+b)+c}) + \epsilon_{(a+b)+c}.
$$

Neglecting the quadratic terms we have:

$$
\tilde{\epsilon}_{(a+b)+c} \approx \frac{a + b}{a + b + c}\epsilon_{a+b} + 1 \cdot \epsilon_{(a+b)+c}.
$$

The amplification factors is 1 for $\epsilon_{(a+b)+c}$ and

$$
\text{Amp}_{(a+b)+c} = \frac{a + b}{a + b + c},
$$

for $\epsilon_{a+b}$. Its size depends on whether $|a + b|$ or $|b + c|$ is the smallest of the two.

The amplification factor for $a + (b + c)$ is

$$
\text{Amp}_{a+(b+c)} = \frac{b + c}{a + b + c}.
$$
For example, if $|b + c| << |a + b|$ then it is better to have $a + (b + c)$.

Examples:

$$a = 1, b = 2, c = 10^7,$$

For Algorithm 1:

$$Amp_{(a+b)+c} = 3/(3 + 10^7) \approx 10^{-7}.$$ 

For Algorithm 2:

$$Amp_{(a+(b+c))} = (2 + 10^7)/(3 + 10^7) \approx 1$$

We remark that Algorithm 1 is numerically more trustworthy than Algorithm 2.

Consider a second numerical example:

$$a = 0.2378912710^5, b = -0.2378955810^5c = 0.001,$$

For Algorithm 1:

$$Amp_{(a+b)+c} = 1.0023,$$

For Algorithm 2:

$$Amp_{a+(b+c)} = (b + c)/(a + b + c) = 55324.6.$$ 

For this example we can see that Algorithm 1: is numerically more trustworthy than Algorithm 2.