FLOATING POINT ARITHMETIC - ERROR ANALYSIS

• Brief review of floating point arithmetic

• Model of floating point arithmetic

• Notation, backward and forward errors
Roundoff errors and floating-point arithmetic

The basic problem: The set $A$ of all possible representable numbers on a given machine is finite - but we would like to use this set to perform standard arithmetic operations (+,*,-,/) on an infinite set. The usual algebra rules are no longer satisfied since results of operations are rounded.

Basic algebra breaks down in floating point arithmetic.

Example: In floating point arithmetic.

$$a + (b + c) \neq (a + b) + c$$

Matlab experiment: For 10,000 random numbers find number of instances when the above is true. Same thing for the multiplication.
**Floating point representation:**

Real numbers are represented in two parts: A mantissa (significand) and an exponent. If the representation is in the base \( \beta \) then:

\[
x = \pm (.d_1 d_2 \cdots d_t) \beta^e
\]

- \( .d_1 d_2 \cdots d_t \) is a fraction in the base-\( \beta \) representation (Generally the form is normalized in that \( d_1 \neq 0 \)), and \( e \) is an integer
- Often, more convenient to rewrite the above as:
  \[
x = \pm (m/\beta^t) \times \beta^e \equiv \pm m \times \beta^{e-t}
\]
- Mantissa \( m \) is an integer with \( 0 \leq m \leq \beta^t - 1 \).
Notation: \( fl(x) \) = closest floating point representation of real number \( x \) ('rounding')

When a number \( x \) is very small, there is a point when \( 1 + x \equiv 1 \) in a machine sense. The computer no longer makes a difference between 1 and \( 1 + x \).

**Machine epsilon:** The smallest number \( \epsilon \) such that \( 1 + \epsilon \) is a float that is different from one, is called machine epsilon. Denoted by \( \text{macheps} \) or \( \text{eps} \), it represents the distance from 1 to the next larger floating point number.

With previous representation, \( \text{eps} \) is equal to \( \beta^{-(t-1)} \).
**Example:** In IEEE standard double precision, $\beta = 2$, and $t = 53$ (includes ‘hidden bit’). Therefore $\epsilon_p = 2^{-52}$.

**Unit Round-off** A real number $x$ can be approximated by a floating number $fl(x)$ with relative error no larger than $u = \frac{1}{2}\beta^{-(t-1)}$.

- $u$ is called Unit Round-off.
- In fact can easily show:

  $$fl(x) = x(1 + \delta) \text{ with } |\delta| < u$$

Matlab experiment: find the machine epsilon on your computer.

- Many discussions on what conditions/ rules should be satisfied by floating point arithmetic. The IEEE standard is a set of standards adopted by many CPU manufacturers.
**Rule 1.**

\[ fl(x) = x(1 + \epsilon), \text{ where } |\epsilon| \leq u \]

**Rule 2.**

For all operations \( \odot \) (one of +, −, *, /)

\[ fl(x \odot y) = (x \odot y)(1 + \epsilon_{\odot}), \text{ where } |\epsilon_{\odot}| \leq u \]

**Rule 3.**

For +, * operations

\[ fl(a \odot b) = fl(b \odot a) \]

\[ \text{Matlab experiment: Verify experimentally Rule 3 with 10,000 randomly generated numbers } a_i, b_i. \]
Example: Consider the sum of 3 numbers: \( y = a + b + c \).

Done as \( fl(fl(a + b) + c) \)

\[
\begin{align*}
\eta &= fl(a + b) = (a + b)(1 + \epsilon_1) \\
y_1 &= fl(\eta + c) = (\eta + c)(1 + \epsilon_2) \\
    &= [(a + b)(1 + \epsilon_1) + c](1 + \epsilon_2) \\
    &= [(a + b + c) + (a + b)\epsilon_1]\(1 + \epsilon_2) \\
    &= (a + b + c)\left[1 + \frac{a + b}{a + b + c}\epsilon_1(1 + \epsilon_2) + \epsilon_2\right]
\end{align*}
\]

So disregarding the high order term \( \epsilon_1\epsilon_2 \)

\[
fl(fl(a + b) + c) = (a + b + c)(1 + \epsilon_3)
\]

\[
\epsilon_3 \approx \frac{a + b}{a + b + c}\epsilon_1 + \epsilon_2
\]
If we redid the computation as $y_2 = \text{fl}(a + \text{fl}(b + c))$ we would find

$$\text{fl}(a + \text{fl}(b + c)) = (a + b + c)(1 + \epsilon_4)$$

$$\epsilon_4 \approx \frac{b + c}{a + b + c} \epsilon_1 + \epsilon_2$$

The error is amplified by the factor $(a + b)/y$ in the first case and $(b + c)/y$ in the second case.

In order to sum $n$ numbers accurately, it is better to start with small numbers first. [However, sorting before adding is not worth it.]

But watch out if the numbers have mixed signs!
**The absolute value notation**

- For a given vector \( \mathbf{x} \), \(|\mathbf{x}|\) is the vector with components \(|x_i|\), i.e., \(|\mathbf{x}|\) is the component-wise absolute value of \( \mathbf{x} \).

- Similarly for matrices:

\[
|A| = \{ |a_{ij}| \}_{i=1,...,m; j=1,...,n}
\]

- An obvious result: The basic inequality

\[
|fl(a_{ij}) - a_{ij}| \leq u |a_{ij}|
\]

translates into

\[
fl(A) = A + E \quad \text{with} \quad |E| \leq u |A|
\]

- \( A \leq B \) means \( a_{ij} \leq b_{ij} \) for all \( 1 \leq i \leq m; 1 \leq j \leq n \)
Backward and forward errors

- Assume the approximation \( \hat{y} \) to \( y = \text{alg}(x) \) is computed by some algorithm with arithmetic precision \( \epsilon \). Possible analysis: find an upper bound for the Forward error

\[ |\Delta y| = |y - \hat{y}| \]

- This is not always easy.

**Alternative question:** find equivalent perturbation on initial data \( x \) that produces the result \( \hat{y} \). In other words, find \( \Delta x \) so that:

\[ \text{alg}(x + \Delta x) = \hat{y} \]

- The value of \( |\Delta x| \) is called the backward error. An analysis to find an upper bound for \( |\Delta x| \) is called Backward error analysis.
Example:

\[
A = \begin{pmatrix} a & b \\ 0 & c \end{pmatrix} \quad B = \begin{pmatrix} d & e \\ 0 & f \end{pmatrix}
\]

Consider the product: \( fl(A.B) = \]

\[
\begin{bmatrix}
ad(1 + \epsilon_1) \\
0
\end{bmatrix}
\begin{bmatrix}
[ae(1 + \epsilon_2) + bf(1 + \epsilon_3)] (1 + \epsilon_4) \\
cf(1 + \epsilon_5)
\end{bmatrix}
\]

with \( \epsilon_i \leq u \), for \( i = 1, \ldots, 5 \). Result can be written as:

\[
\begin{bmatrix}
a \\
0
\end{bmatrix}
\begin{bmatrix}
b(1 + \epsilon_3)(1 + \epsilon_4) \\
c(1 + \epsilon_5)
\end{bmatrix}
\begin{bmatrix}
d(1 + \epsilon_1) \\
0
\end{bmatrix}
\begin{bmatrix}
e(1 + \epsilon_2)(1 + \epsilon_4) \\
f
\end{bmatrix}
\]

\[ \Rightarrow \]

So \( fl(A.B) = (A + E_A)(B + E_B) \).

\[ \Rightarrow \]

Backward errors \( E_A, E_B \) satisfy:

\( |E_A| \leq 2u |A| + O(u^2) \); \( |E_B| \leq 2u |B| + O(u^2) \)
When solving $Ax = b$ by Gaussian Elimination, we will see that a bound on $\|e_x\|$ such that this holds exactly:

$$A(x_{\text{computed}} + e_x) = b$$

is much harder to find than bounds on $\|E_A\|$, $\|e_b\|$ such that this holds exactly:

$$(A + E_A)x_{\text{computed}} = (b + e_b).$$

**Note:** In many instances backward errors are more meaningful than forward errors: if initial data is accurate only to 4 digits say, then my algorithm for computing $x$ need not guarantee a backward error of less then $10^{-10}$ for example. A backward error of order $10^{-4}$ is acceptable.
**Error Analysis: Inner product**

- Inner products are in the innermost parts of many calculations. Their analysis is important.

**Lemma:** If $|\delta_i| \leq u$ and $nu < 1$ then

$$\prod_{i=1}^{n}(1 + \delta_i) = 1 + \theta_n \quad \text{where} \quad |\theta_n| \leq \frac{nu}{1 - nu}$$

- Common notation $\gamma_n \equiv \frac{nu}{1-nu}$

Prove the lemma [Hint: use induction]
Can use the following simpler result:

**Lemma:** If $|\delta_i| \leq u$ and $nu < .01$ then

$$\Pi_{i=1}^{n} (1 + \delta_i) = 1 + \theta_n \quad \text{where} \quad |\theta_n| \leq 1.01nu$$

**Example:** Previous sum of numbers can be written

$$fl(a + b + c) = a(1 + \epsilon_1)(1 + \epsilon_2)$$
$$+ b(1 + \epsilon_1)(1 + \epsilon_2) + c(1 + \epsilon_2)$$
$$= a(1 + \theta_1) + b(1 + \theta_2) + c(1 + \theta_3)$$
$$= \text{exact sum of slightly perturbed inputs},$$

where all $\theta_i$’s satisfy $|\theta_i| \leq 1.01nu$ (here $n = 2$).

Alternatively, can write ‘forward’ bound:

$$|fl(a + b + c) - (a + b + c)| \leq |a\theta_1| + |b\theta_2| + |c\theta_3|.$$
Consider 

\[ s_n = fl(x_1 * y_1 + x_2 * y_2 + \cdots + x_n * y_n) \]

- In what follows \( \eta_i \)'s come from \( * \), \( \epsilon_i \)'s come from \( + \)
- They satisfy: \( |\eta_i| \leq u \) and \( |\epsilon_i| \leq u \).
- The inner product \( s_n \) is computed as:

1. \( s_1 = fl(x_1 y_1) = (x_1 y_1)(1 + \eta_1) \)
2. \( s_2 = fl(s_1 + fl(x_2 y_2)) = fl(s_1 + x_2 y_2(1 + \eta_2)) = (x_1 y_1(1 + \eta_1) + x_2 y_2(1 + \eta_2))(1 + \epsilon_2) = x_1 y_1(1 + \eta_1)(1 + \epsilon_2) + x_2 y_2(1 + \eta_2)(1 + \epsilon_2) \)
3. \( s_3 = fl(s_2 + fl(x_3 y_3)) = fl(s_2 + x_3 y_3(1 + \eta_3)) = (s_2 + x_3 y_3(1 + \eta_3))(1 + \epsilon_3) \)
Expand:  \( s_3 = x_1 y_1 (1 + \eta_1)(1 + \epsilon_2)(1 + \epsilon_3) + x_2 y_2 (1 + \eta_2)(1 + \epsilon_2)(1 + \epsilon_3) + x_3 y_3 (1 + \eta_3)(1 + \epsilon_3) \)

\[ \text{Induction would show that [with convention that } \epsilon_1 \equiv 0 \text{]} \]

\[ s_n = \sum_{i=1}^{n} x_i y_i (1 + \eta_i) \prod_{j=i}^{n} (1 + \epsilon_j) \]

**Q:** How many terms in the coefficient of \( x_i y_i \) do we have?

- When \( i > 1 \): \( 1 + (n - i + 1) = n - i + 2 \)
- When \( i = 1 \): \( n \) (since \( \epsilon_1 = 0 \) does not count)

**A:** Bottom line: always \( \leq n \).
For each of these products

\[(1 + \eta_i) \prod_{j=i}^{n}(1 + \epsilon_j) = 1 + \theta_i, \quad \text{with} \quad |\theta_i| \leq \gamma_n u\]

so:

\[s_n = \sum_{i=1}^{n} x_i y_i (1 + \theta_i) \quad \text{with} \quad |\theta_i| \leq \gamma_n \]
or:

\[
fl \left( \sum_{i=1}^{n} x_i y_i \right) = \sum_{i=1}^{n} x_i y_i + \sum_{i=1}^{n} x_i y_i \theta_i \quad \text{with} \quad |\theta_i| \leq \gamma_n
\]

This leads to the final result (forward form)

\[
\left| fl \left( \sum_{i=1}^{n} x_i y_i \right) - \sum_{i=1}^{n} x_i y_i \right| \leq \gamma_n \sum_{i=1}^{n} |x_i||y_i|
\]

or (backward form)

\[
fl \left( \sum_{i=1}^{n} x_i y_i \right) = \sum_{i=1}^{n} x_i y_i (1 + \theta_i) \quad \text{with} \quad |\theta_i| \leq \gamma_n
\]
Main result on inner products:

- Backward error expression:
  \[
  fl(x^T y) = [x \cdot (1 + d_x)]^T[y \cdot (1 + d_y)]
  \]
  where \( \|d_\Box\|_\infty \leq 1.01nu \), \( \Box = x, y \).

- Can show equality valid even if one of the \( d_x, d_y \) absent.

- Forward error expression:
  \[
  |fl(x^T y) - x^T y| \leq \gamma_n |x|^T |y|
  \]
  with \( 0 \leq \gamma_n \leq 1.01nu \).

- Elementwise absolute value \( |x| \) and multiply \( \cdot \) notation.

- Above assumes \( nu \leq .01 \).
  For \( u = 2.0 \times 10^{-16} \), this holds for \( n \leq 4.5 \times 10^{13} \).
Consequence of lemma for matrix products:

\[ |fl(AB) - AB| \leq \gamma_n |A||B| \]

Another way to write the result (less precise) is

\[ |fl(x^T y) - x^T y| \leq n \ u \ |x|^T |y| + O(u^2) \]
Assume you use single precision for which you have $u = 2 \times 10^{-6}$. What is the largest $n$ for which $nu \leq 0.01$ holds? Any conclusions for the use of single precision arithmetic?

What does the main result on inner products imply for the case when $y = x$? [Contrast the relative accuracy you get in this case vs. the general case when $y \neq x$]
Show for any $x, y$, there exist $\Delta x, \Delta y$ such that

\[
fl(x^T y) = (x + \Delta x)^T y, \quad \text{with} \quad |\Delta x| \leq \gamma_n |x|
\]
\[
fl(x^T y) = x^T (y + \Delta y), \quad \text{with} \quad |\Delta y| \leq \gamma_n |y|
\]

(Continuation) Let $A$ an $m \times n$ matrix, $x$ an $n$-vector, and $y = Ax$. Show that there exist a matrix $\Delta A$ such

\[
fl(y) = (A + \Delta A)x, \quad \text{with} \quad |\Delta A| \leq \gamma_n |A|
\]

(Continuation) From the above derive a result about a column of the product of two matrices $A$ and $B$. Does a similar result hold for the product $AB$ as a whole?
Error Analysis for linear systems: Triangular case

Recall

**ALGORITHM : 1. Back-Substitution algorithm**

\[
\begin{align*}
&\text{For } i = n : -1 : 1 \text{ do:} \\
&t := b_i \\
&\text{For } j = i + 1 : n \text{ do} \\
&t := t - a_{ij}x_j \\
&\text{End} \\
&x_i = t / a_{ii} \\
&\text{End}
\end{align*}
\]

We must require that each \( a_{ii} \neq 0 \)

Round-off error (use previous results for \((\cdot, \cdot)\))?
The computed solution $\hat{x}$ of the triangular system $Ux = b$ computed by the back-substitution algorithm satisfies:

$$(U + E)\hat{x} = b$$

with

$$|E| \leq n \ u \ |U| + O(u^2)$$

➤ **Backward error analysis.** Computed $x$ solves a slightly perturbed system.

➤ **Backward error not large in general.** It is said that triangular solve is “backward stable”.
Error Analysis for Gaussian Elimination

If no zero pivots are encountered during Gaussian elimination (no pivoting) then the computed factors \( \hat{L} \) and \( \hat{U} \) satisfy

\[
\hat{L}\hat{U} = A + H
\]

with

\[
|H| \leq 3(n - 1) \times u \left(|A| + |\hat{L}| |\hat{U}|\right) + O(u^2)
\]

Solution \( \hat{x} \) computed via \( \hat{L}\hat{y} = b \) and \( \hat{U}\hat{x} = \hat{y} \) is s. t.

\[
(A + E)\hat{x} = b \text{ with}
\]

\[
|E| \leq nu \left(3|A| + 5 |\hat{L}| |\hat{U}|\right) + O(u^2)
\]
“Backward” error estimate.

| | | are not known in advance – they can be large.

What if partial pivoting is used?

Permutations introduce no errors. Equivalent to standard LU factorization on matrix $PA$.

| is small since $l_{ij} \leq 1$. Therefore, only $U$ is “uncertain”

In practice partial pivoting is “stable” – i.e., it is highly unlikely to have a very large $U$. 
Supplemental notes: Floating Point Arithmetic

In most computing systems, real numbers are represented in two parts: A mantissa and an exponent. If the representation is in the base $\beta$ then:

$$x = \pm (.d_1 d_2 \cdots d_m) \beta^e$$

- $d_1 d_2 \cdots d_m$ is a fraction in the base-$\beta$ representation.
- $e$ is an integer - can be negative, positive or zero.
- Generally the form is normalized in that $d_1 \neq 0$. 
Example: In base 10 (for illustration)

1. 1000.12345 can be written as
   \[0.100012345_{10} \times 10^4\]

2. 0.000812345 can be written as
   \[0.812345_{10} \times 10^{-3}\]

Problem with floating point arithmetic: we have to live with limited precision.

Example: Assume that we have only 5 digits of accuracy in the mantissa and 2 digits for the exponent (excluding sign).

\[.d_1 \, d_2 \, d_3 \, d_4 \, d_5 \, e_1 \, e_2\]
Try to add $1000.2 = .10002e+03$ and $1.07 = .10700e+01$:

$$1000.2 = \boxed{.1\,0\,0\,0\,2\,0\,4} ; \quad 1.07 = \boxed{.1\,0\,7\,0\,0\,0\,1}$$

**First task:** align decimal points. The one with smallest exponent will be (internally) rewritten so its exponent matches the largest one:

$$1.07 = 0.000107 \times 10^4$$

**Second task:** add mantissas:

$$\begin{align*}
0.10002 & \\
+ 0.000107 & \\
\hline
= 0.100127 &
\end{align*}$$
**Third task:**
round result. Result has 6 digits - can use only 5 so we can

- Chop result: \(0.10012\);
- Round result: \(0.10013\);

**Fourth task:**
Normalize result if needed (not needed here)

result with rounding: \(0.1001304\);

Redo the same thing with 7000.2 + 4000.3 or 6999.2 + 4000.3.
The IEEE standard

32 bit (Single precision):

<table>
<thead>
<tr>
<th>± 8 bits</th>
<th>← 23 bits →</th>
</tr>
</thead>
<tbody>
<tr>
<td>sign</td>
<td>exponent</td>
</tr>
</tbody>
</table>

- Number is scaled so it is in the form $1.d_1d_2...d_{23} \times 2^e$ - but leading one is not represented.

- $e$ is between -126 and 127.

- [Here is why: Internally, exponent $e$ is represented in “biased” form: what is stored is actually $c = e + 127$ - so the value $c$ of exponent field is between 1 and 254. The values $c = 0$ and $c = 255$ are for special cases (0 and $\infty$)]
64 bit (Double precision):

<table>
<thead>
<tr>
<th>±</th>
<th>11 bits</th>
<th>←</th>
<th>52 bits</th>
<th>→</th>
</tr>
</thead>
<tbody>
<tr>
<td>sign</td>
<td>exponent</td>
<td></td>
<td>mantissa</td>
<td></td>
</tr>
</tbody>
</table>

- Bias of 1023 so if $e$ is the actual exponent the content of the exponent field is $c = e + 1023$
- Largest exponent: $1023$; Smallest = $-1022$.
- $c = 0$ and $c = 2047$ (all ones) are again for 0 and $\infty$
- Including the hidden bit, mantissa has total of 53 bits (52 bits represented, one hidden).
- In single precision, mantissa has total of 24 bits (23 bits represented, one hidden).
Take the number 1.0 and see what will happen if you add \(1/2, 1/4, \ldots, 2^{-i}\). Do not forget the hidden bit!

![Hidden bit diagram](image)

(Note: The ‘e’ part has 12 bits and includes the sign)

**Conclusion**

\[ \text{fl}(1 + 2^{-52}) \neq 1 \text{ but: } \text{fl}(1 + 2^{-53}) == 1 !! \]
Special Values

- Exponent field = 00000000000 (smallest possible value)
  No hidden bit. All bits == 0 means exactly zero.

- Allow for unnormalized numbers,
  leading to gradual underflow.

- Exponent field = 11111111111 (largest possible value)
  Number represented is ”Inf” ”-Inf” or ”NaN”.
Recent trend: GPUs

- Graphics Processor Units: Very fast boards attached to CPUs for heavy-duty computing
- e.g., NVIDIA V100 can deliver 112 Teraflops (1 Teraflops = $10^{12}$ operations per second) for certain types of computations.
- Single precision much faster than double ...
- ... and there is also “half-precision” which is $\approx 16$ times faster than standard 64bit arithmetic
- Used primarily for Deep-learning