3

Errors in computation; where do they come from?

L. Olson

September 1, 2015

Department of Computer Science
University of Illinois at Urbana-Champaign

objectives

- look at floating point representation in its basic form
- expose errors of a different form: rounding error
- highlight IEEE-754 standard

why this is important:

- Errors come in two forms: truncation error and rounding error
  - we always have them . . .
  - case study: Intel
  - our jobs as developers: reduce impact

example: calculating $x = x + 0.1$

next: floating point numbers

- We're familiar with base 10 representation of numbers:

$$1234 = 4 \times 10^0 + 3 \times 10^1 + 2 \times 10^2 + 1 \times 10^3$$

and

$$.1234 = 1 \times 10^{-1} + 2 \times 10^{-2} + 3 \times 10^{-3} + 4 \times 10^{-4}$$

- we write 1234.1234 as an integer part and a fractional part:

$$a_3 a_2 a_1 a_0 . b_1 b_2 b_3 b_4$$

- For some (even simple) numbers, there may be an *infinite* number of digits to the right:

$$\pi = 3.14159\ldots$$
$$1/9 = 0.11111\ldots$$
$$\sqrt{2} = 1.41421\ldots$$

other bases

- So far, we have just base 10. What about base β?
- binary ($\beta = 2$), octal ($\beta = 8$), hexadecimal ($\beta = 16$), etc
- In the β-system we have

$$(a_n \dots a_2 a_1 a_0 . b_1 b_2 b_3 b_4 \dots)_\beta = \sum_{k=0}^{n} a_k \beta^k + \sum_{k=0}^{\infty} b_k \beta^{-k}$$

integer conversion

An algorithm to compute the base 2 representation of a base 10 integer

$$\begin{aligned}(N)_{10} &= (a_j a_{j-1} \dots a_2 a_0)_2 \\ &= a_j \cdot 2^j + \dots + a_1 \cdot 2^1 + a_0 \cdot 2^0\end{aligned}$$

Compute $(N)_{10}/2 = Q + R/2$:

$$\frac{N}{2} = \underbrace{a_j \cdot 2^{j-1} + \dots + a_1 \cdot 2^0}_{=Q} + \underbrace{\frac{a_0}{2}}_{=R/2}$$

Example

Example: compute $(11)_{10}$ base 2

$$\begin{aligned}11/2 = 5R1 \quad &\Rightarrow \quad a_0 = 1 \\ 5/2 = 2R1 \quad &\Rightarrow \quad a_1 = 1 \\ 2/2 = 1R0 \quad &\Rightarrow \quad a_2 = 0 \\ 1/2 = 0R1 \quad &\Rightarrow \quad a_3 = 1\end{aligned}$$

So $(11)_{10} = (1011)_2$

the other way...

Convert a base-2 number to base-10:

$$
\begin{aligned}
&(11\,000\,101)_2 \\
&= 1 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\
&= 1 + 2(0 + 2(1 + 2(0 + 2(0 + 2(0 + 2(1 + 2(1)))))))) \\
&= 197
\end{aligned}
$$

converting fractions

- straight forward way is not easy
- goal: for $x \in [0, 1]$ write

$$x = 0.b_1 b_2 b_3 b_4 \cdots = \sum_{k=1}^{\infty} c_k \beta^{-k} = (0.c_1 c_2 c_3 \dots)_\beta$$

- $\beta(x) = (c_1.c_2 c_3 c_4 \dots)_\beta$
- multiplication by β in base-β only shifts the radix

fraction algorithm

An algorithm to compute the binary representation of a fraction x:

$$\begin{aligned} x &= 0.b_1 b_2 b_3 b_4 \ldots \\ &= b_1 \cdot 2^{-1} + \ldots \end{aligned}$$

Multiply x by 2. The integer part of $2x$ is b_1

$$2x = b_1 \cdot 2^0 + b_2 \cdot 2^{-1} + b_3 \cdot 2^{-2} + \ldots$$

Example

Example:Compute the binary representation of 0.625

$$\begin{aligned} 2 \cdot 0.625 = 1.25 \quad &\Rightarrow \quad b_{-1} = 1 \\ 2 \cdot 0.25 = 0.5 \quad &\Rightarrow \quad b_{-2} = 0 \\ 2 \cdot 0.5 = 1.0 \quad &\Rightarrow \quad b_{-3} = 1 \end{aligned}$$

So $(0.625)_{10} = (0.101)_2$

a problem with precision

```
1 r_0 = x
2 for  k = 1, 2, ..., m
3    if  r_{k-1} ≥ 2^{-k}
4       b_k = 1
5       r_k = r_{k-1} − 2^{-k}
6    else
7       b_k = 0
8    end
9 end
```

k	2^{-k}	b_k	$r_k = r_{k-1} - b_k 2^{-k}$
0			0.8125
1	0.5	1	0.3125
2	0.25	1	0.0625
3	0.125	0	0.0625
4	0.0625	1	0.0000

binary fraction example

a problem with precision

For other numbers, such as $\frac{1}{5} = 0.2$, an infinite length is needed.

$0.2 \quad \rightarrow \quad .0011\,0011\,0011\,\ldots$

So 0.2 is stored just fine in base-10, but needs infinite number of digits in base-2

!!!

This is *roundoff* error in its basic form...

Business Day

The New York Times

D1

THURSDAY, NOVEMBER 24, 1994

BUSINESS Digest

THURSDAY, NOVEMBER 24, 1994

DOW 30 Industrials	DOLLAR vs. Japanese Yen	OIL Nymex Spot	BONDS 30-Year Treasuries
3,674.63 −3.36	98.47 Yen +0.24 Yen	$18.15 +$0.33	7.94% −0.08

Companies

A flaw in the Pentium, the top chip made by Intel, can cause inaccurate calculations in certain rare cases. Intel said the chip would not have to be recalled. But many scientists and engineers who depend on precise calculations are concerned. [Page D1.]

Gibson Greetings and Bankers Trust settled a lawsuit over derivatives. Gibson agreed to pay Bankers Trust nearly $6.2 million, or about 30 percent of the $20.7 million that Bankers claimed it was owed in the trades between the two. [D1.]

Sorry, Charlie: H. J. Heinz and Leo Burnett are parting ways after 27 years. The Burnett agency created familiar advertising characters like Morris the finicky cat and Charlie the tuna. [D1.]

Pfizer will buy SmithKline Beecham's animal health business for $1.45 billion in cash. The deal would make Pfizer the world's largest maker of animal drugs, topping Merck. [D4.]

Kmart is seeking ways to save as much as $800 million in two years in addition to store closings and job cuts. [D4.]

A Federal judge froze a New York brokerage account of a British businessman accused of illegal insider trading. [D4.]

The F.T.C. is seeking data on alliances with Caremark International, in a broadening of an investigation of relations between drug makers and managed health-care drug distributors. [D4.]

Metallgesellschaft of Germany plans to write down the paper value of its shares by half in an effort to rescue itself. [D4.]

I.B.M., AT&T, Apple Computer and Siemens plan to create a universal communication language for computers. [D5.]

Markets

After a roller-coaster session, stocks ended slightly lower. The Standard & Poor's 500 index fell just 0.16 point, easing investors' concerns that stocks were in for a long, hard descent. [D1.]

A stock slide is not likely to change Federal Reserve policy. Indeed, stocks' strength until now surprised Fed officials. [D8.]

Stock markets fell sharply across Europe as investors took their cue from Wall Street and sought refuge in bonds. [D8.]

Flaw Undermines Accuracy of Pentium Chips

By JOHN MARKOFF

Special to The New York Times

SAN FRANCISCO, Nov. 23 — An elusive circuitry error is causing a chip used in millions of computers to generate inaccurate results in certain rare cases, heightening anxiety among many scientists and engineers who rely on their machines for precise calculations.

The flaw, an error in division, has been found in the Pentium, the current top microprocessor of the Intel Corporation, the world's largest chip maker. The chip, in several different configurations, is used in many computers sold for home and business use, including those made by I.B.M., Compaq, Dell, Gateway 2000 and others.

The flaw appears in all Pentium chips now on the market, in certain types of division problems involving more than five significant digits, a mathematical term that can include numbers before and after a decimal point.

Intel declined to say how many Pentium chips it made or sold, but Dataquest, a market research company in San Jose, Calif., estimated that in 1994 Intel would sell 5.5 million to 6 million Pentiums, roughly 10 percent of the number of personal computers sold worldwide.

Intel said yesterday that it did not believe the chip needed to be recalled, asserting that the typical user would have but one chance in more than nine billion of encountering an inaccurate result as a consequence of the error, and thus there was no noticeable consequence to users of business or home computers. Indeed, the company said it was continuing to send computer makers Pentium chips built before the problem was detected.

William Kahan of the University of California at Berkeley, one of the nation's experts on computer mathematics, expressed skepticism about Intel's contentions that the error would only occur in extremely rare instances.

"These kinds of statistics have to cause some wonderment," he said. "They are based on assertions about the probability of events whose probability we don't know."

At the Jet Propulsion Laboratory in Pasadena, Calif., one satellite communications researcher who learned of the error this week, said six Pentium machines were used in his group and their use had been suspended for now.

"The Pentium appeared as a cost-

Continued on Page D5

Four Sharp Corrections in the Dow

This year the stock market has tended to gain gradually for weeks or bob nervously around a trading range with little direction, waiting for the moment when some bad news or unfavorable trend sets off a headlong flight. Then, once prices have been beaten down for a few days, the buyers reappear and the bleeding stops. Four of these corrections are highlighted, along with the development that traders said pushed the market over the precipice.

Gibson Suit On Trades Is Settled

Bankers Trust Gets 30% of Debt Claimed

By MICHAEL QUINT

Gibson Greetings Inc. and the Bankers Trust Company said yesterday that they had settled Gibson's lawsuit accusing the bank of improperly leading the company to engage in risky financial trades.

Under the out-of-court settlement, Gibson will pay Bankers Trust nearly $6.2 million, or about 30 percent of the $20.7 million that Bankers Trust contended it was owed under the trades.

While neither side boasted of victory, Douglas Kidd, a spokesman for Bankers Trust, acknowledged that

Flaw Undermines Accuracy of Pentium Chips
By JOHN MARKOFFSpecial to The New York Times
New York Times (1857-Current file); Nov 24, 1994; ProQuest Historical Newspapers The New York Times (1851 - 2002)
pg. D1

Flaw Undermines Accuracy of Pentium Chips

By JOHN MARKOFF

Special to The New York Times

SAN FRANCISCO, Nov. 23 — An elusive circuitry error is causing a chip used in millions of computers to generate inaccurate results in certain rare cases, heightening anxiety among many scientists and engineers who rely on their machines for precise calculations.

The flaw, an error in division, has been found in the Pentium, the current top microprocessor of the Intel Corporation, the world's largest chip maker. The chip, in several different configurations, is used in many computers sold for home and business use, including those made by I.B.M., Compaq, Dell, Gateway 2000 and others.

The flaw appears in all Pentium chips now on the market, in certain types of division problems involving more than five significant digits, a mathematical term that can include numbers before and after a decimal point.

Intel declined to say how many Pentium chips it made or sold, but Dataquest, a market research company in San Jose, Calif., estimated that in 1994 Intel would sell 5.5 million to 6 million Pentiums, roughly 10 percent of the number of personal computers sold worldwide.

Intel said yesterday that it did not believe the chip needed to be recalled, asserting that the typical user would have but one chance in more than nine billion of encountering an inaccurate result as a consequence of the error, and thus there was no noticeable consequence to users of business or home computers. Indeed, the company said it was continuing to send computer makers Pentium chips built before the problem was detected.

William Kahan of the University of California at Berkeley, one of the nation's experts on computer mathematics, expressed skepticism about Intel's contentions that the error would only occur in extremely rare instances.

"These kinds of statistics have to cause some wonderment," he said. "They are based on assertions about the probability of events whose probability we don't know."

At the Jet Propulsion Laboratory in Pasadena, Calif., one satellite communications researcher who learned of the error this week, said six Pentium machines were used in his group and their use had been suspended for now.

"The Pentium appeared as a cost-

Continued on Page D5

tium Chips

In some complex division problems, annoying errors.

corrected.

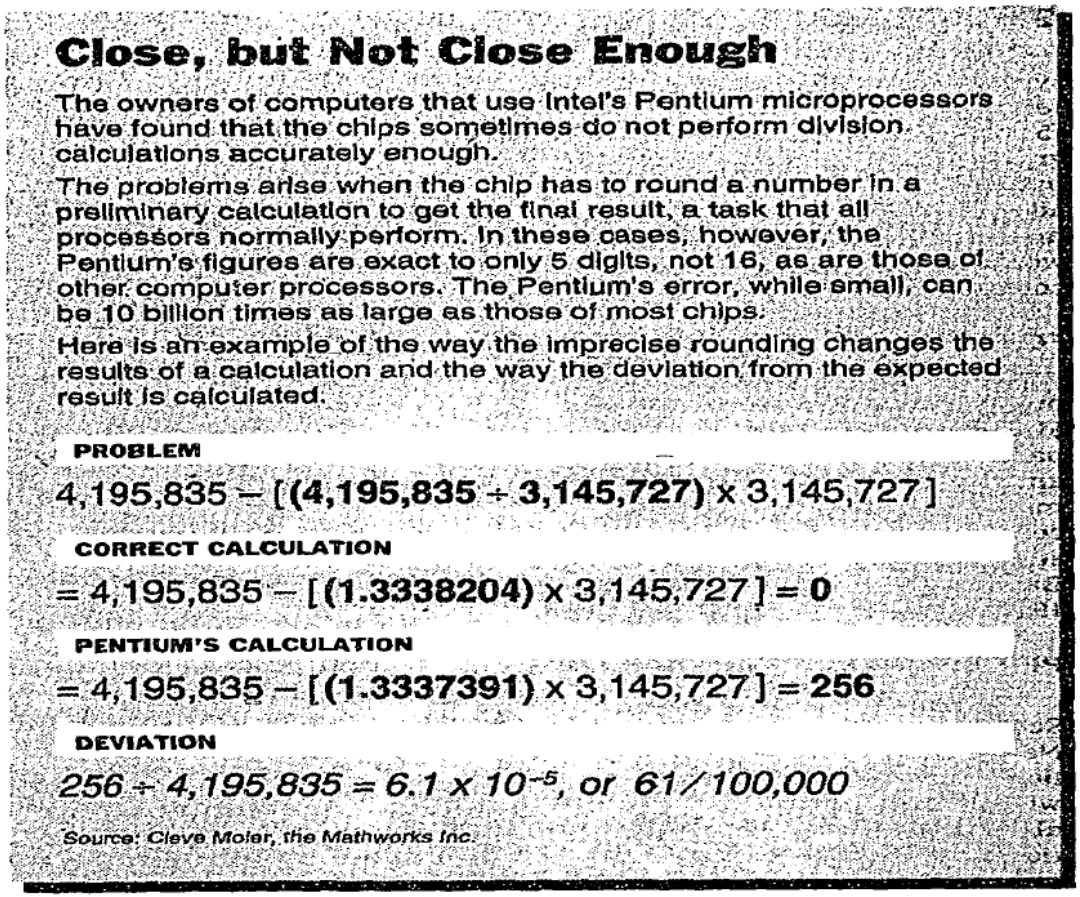
Some computer users said they believed that Intel had not acted quickly enough after discovering the error.

"Intel has known about this since the summer; why didn't they tell anyone?" said Andrew Schulman, the author of a series of technical books on PC's. "It's a hot issue, and I don't think they've handled this well.

The company said that after it discovered the problem this summer, it ran months of simulations of different applications, with the help of outside experts, to determine whether the problem was serious.

The Pentium error occurs in a portion of the chip known as the floating point unit, which is used for extremely precise computations. In rare cases, the error shows up in the result of a division operation.

Intel said the error occurred because of an omission in the translation of a formula into computer



Close, but Not Close Enough

The owners of computers that use Intel's Pentium microprocessors have found that the chips sometimes do not perform division calculations accurately enough.

The problems arise when the chip has to round a number in a preliminary calculation to get the final result, a task that all processors normally perform. In these cases, however, the Pentium's figures are exact to only 5 digits, not 16, as are those of other computer processors. The Pentium's error, while small, can be 10 billion times as large as those of most chips.

Here is an example of the way the imprecise rounding changes the results of a calculation and the way the deviation from the expected result is calculated.

PROBLEM

$4,195,835 - [(4,195,835 \div 3,145,727) \times 3,145,727]$

CORRECT CALCULATION

$= 4,195,835 - [(1.3338204) \times 3,145,727] = 0$

PENTIUM'S CALCULATION

$= 4,195,835 - [(1.3337391) \times 3,145,727] = 256$

DEVIATION

$256 \div 4,195,835 = 6.1 \times 10^{-5}$, or 61/100,000

Source: Cleve Moler, the Mathworks Inc.

intel timeline

June 1994	Intel engineers discover the division error. Managers decide the error will not impact many people. Keep the issue internal.
June 1994	Dr Nicely at Lynchburg College notices computation problems
Oct 19, 1994	After months of testing, Nicely confirms that other errors are not the cause. The problem is in the Intel Processor.
Oct 24, 1994	Nicely contacts Intel. Intel duplicates error.
Oct 30, 1994	After no action from Intel, Nicely sends an email

intel timeline

```
FROM:  Dr. Thomas R. Nicely
       Professor of Mathematics
       Lynchburg College
       1501 Lakeside Drive
       Lynchburg, Virginia 24501-3199

       Phone:     804-522-8374
       Fax:       804-522-8499
       Internet:  nicely@acavax.lynchburg.edu

TO:    Whom it may concern

RE:    Bug in the Pentium FPU

DATE:  30 October 1994

It appears that there is a bug in the floating point unit
(numeric coprocessor) of many, and perhaps all, Pentium
processors.

In short, the Pentium FPU is returning erroneous values
for certain division operations. For example,
     0001/824633702441.0
is calculated incorrectly (all digits beyond the eighth
significant digit are in error). This can be verified in
compiled code, an ordinary spreadsheet such as Quattro Pro
or Excel, or even the Windows calculator (use the
scientific mode), by computing
     00(824633702441.0)*(1/824633702441.0),
which should equal 1 exactly (within some extremely small
rounding error; in general, coprocessor results should
contain 19 significant decimal digits).  However, the
Pentiums tested return
     0000.999999996274709702
.
.
.
```

intel timeline

Nov 1, 1994	Software company Phar Lap Software receives Nicely's email. Sends to colleagues at Microsoft, Borland, Watcom, etc. decide the error will not impact many people. Keep the issue internal.
Nov 2, 1994	Email with description goes global.
Nov 15, 1994	USC reverse-engineers the chip to expose the problem. Intel still denies a problem. Stock falls.
Nov 22, 1994	CNN *Moneyline* interviews Intel. Says the problem is minor.
Nov 23, 1994	The MathWorks develops a fix.
Nov 24, 1994	New York Times story. Intel still sending out flawed chips. Will replace chips only if it caused a problem in an important application.
Dec 12, 1994	IBM halts shipment of Pentium based PCs
Dec 16, 1994	Intel stock falls again.
Dec 19, 1994	More reports in the NYT: lawsuits, etc.
Dec 20, 1994	Intel admits. Sets aside $420 million to fix.

numerical "bugs"

Obvious

Software has bugs

Not-SO-Obvious

Numerical software has two unique bugs:

1. roundoff error
2. truncation error

numerical errors

Roundoff

Roundoff occurs when digits in a decimal point (0.3333...) are lost (0.3333) due to a limit on the memory available for storing one numerical value.

Truncation

Truncation error occurs when discrete values are used to approximate a mathematical expression.

uncertainty: well- or ill-conditioned?

Errors in input data can cause *uncertain* results

- input data can be experimental or rounded. leads to a certain variation in the results
- *well-conditioned*: numerical results are insensitive to small variations in the input
- *ill-conditioned*: small variations lead to drastically different numerical calculations (a.k.a. poorly conditioned)

our job

As numerical analysts, we need to

1. solve a problem so that the calculation is not susceptible to large roundoff error
2. solve a problem so that the approximation has a *tolerable* truncation error

How?

- incorporate roundoff-truncation knowledge into
  - the mathematical model
  - the method
  - the algorithm
  - the software design
- awareness $\rightarrow$ correct interpretation of results

floating points

Normalized Floating-Point Representation

Real numbers are stored as

$$x = \pm(0.d_1 d_2 d_3 \ldots d_m)_\beta \times \beta^e$$

- $d_1 d_2 d_3 \ldots d_m$ is the mantissa, e is the exponent
- e is negative, positive or zero
- the general normalized form requires $d_1 \neq 0$

floating point

Example

In base 10

- 1000.12345 can be written as

$$(0.100012345)_{10} \times 10^4$$

- 0.000812345 can be written as

$$(0.812345)_{10} \times 10^{-3}$$

floating point

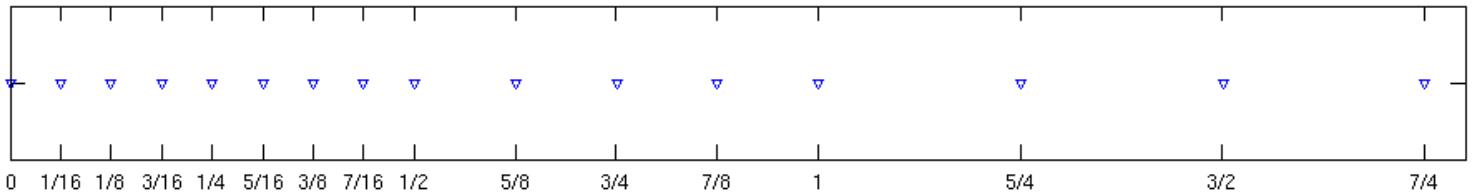
Suppose we have only 3 bits for a mantissa and a 1 bit exponent stored like

$.d_1$	d_2	d_3	e_1

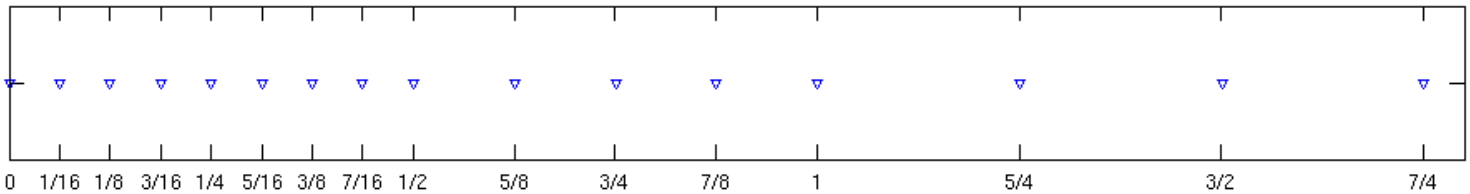
All possible combinations give:

$$\begin{aligned} 000_2 &= 0 \\ &\dots \qquad \times 2^{-1,0,1} \\ 111_2 &= 7 \end{aligned}$$

So we get $0, \frac{1}{16}, \frac{2}{16}, \dots, \frac{7}{16}$, $0, \frac{1}{4}, \frac{2}{4}, \dots, \frac{7}{4}$, and $0, \frac{1}{8}, \frac{2}{8}, \dots, \frac{7}{8}$. On the real line:



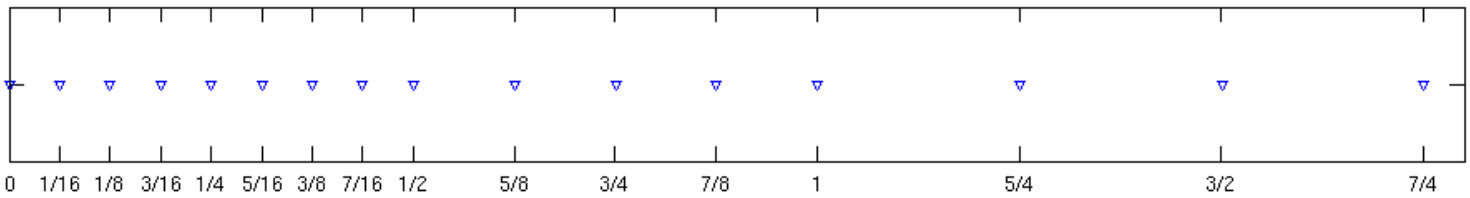
overflow, underflow

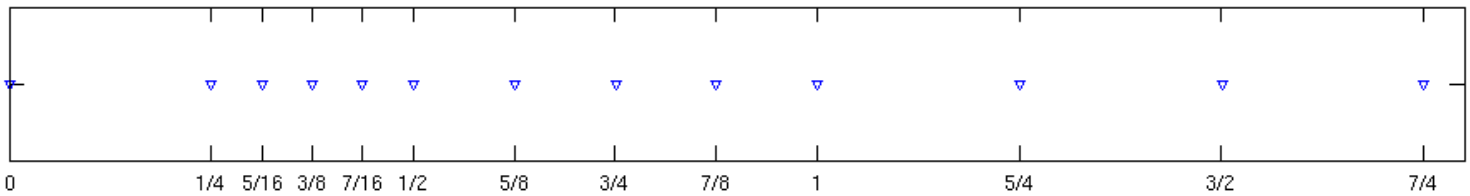


- computations too close to zero may result in *underflow*
- computations too large may result in *overflow*
- overflow error is considered more severe
- underflow can just fall back to 0

normalizing

If we use the normalized form in our 4-bit case, we lose $0.001_2 \times 2^{-1,0,1}$ along with other. So we cannot represent $\frac{1}{16}$, $\frac{1}{8}$, and $\frac{3}{16}$.



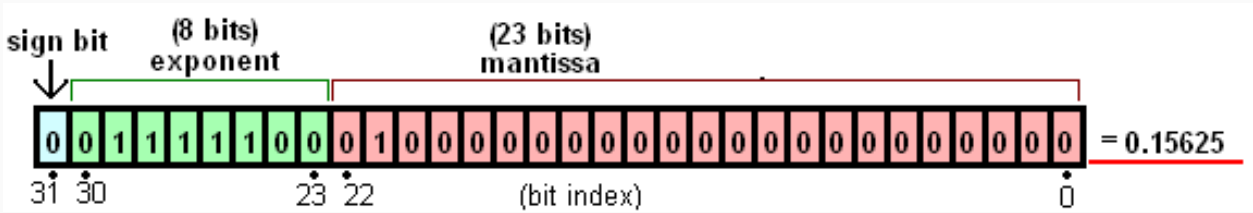


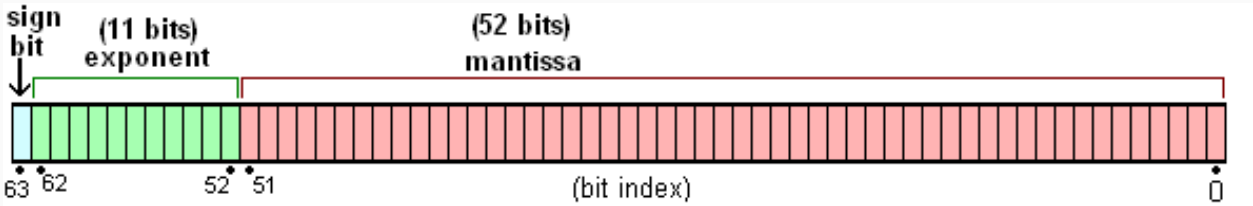
ieee-754 why this is important:

- IEEE-754 is a widely used standard accepted by hardware/software makers
  - defines the floating point distribution for our computation
  - offer several rounding modes which effect accuracy
- Floating point arithmetic emerges in nearly every piece of code
  - even modest mathematical operation yield loss of significant bits
  - several pitfalls in common mathematical expressions

ieee floating point (v. 754)

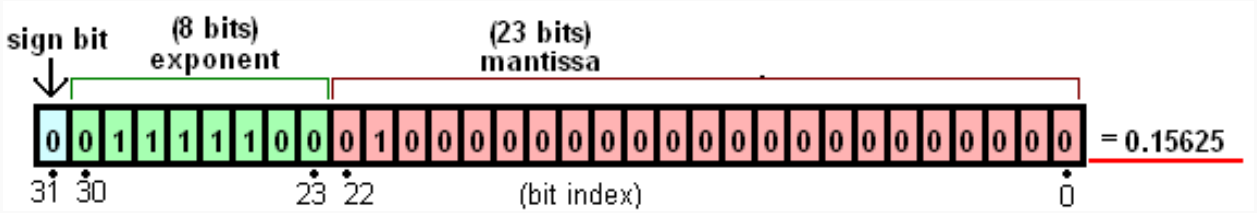
- How much storage do we actually use in practice?
- 32-bit word lengths are typical
- IEEE Single-precision floating-point numbers use 32 bits
- IEEE Double-precision floating-point numbers use 64 bits
- Goal: use the 32-bits to best represent the normalized floating point number





ieee single precision (marc-32)

$$x = \pm q \times 2^m$$



Notes:

- 1-bit sign
- 8-bit exponent $|m|$
- 23-bit mantissa q
- The leading one in the mantissa q does not need to be represented: $b_1 = 1$ is hidden bit
- IEEE 754: put x in $1.f$ normalized form
- $0 < m + 127 = c < 255$
- Largest exponent $= 127$, Smallest exponent $= -126$
- Special cases: $c = 0, 255$

ieee single precision

$$x = \pm q \times 2^m$$

Process for $x = -52.125$:

1. Convert both integer and fractional to binary:
   $x = -(110100.00100000000)_2$
2. Convert to $1.f$ form: $x = \underbrace{-}_{1}(1.\underbrace{101\,000\,010\,000\,\ldots\,0}_{23})_2 \times 2^5$
3. Convert exponent $5 = c - 127 \Rightarrow c = 132 \Rightarrow c = (\underbrace{10\,000\,100}_{8})_2$

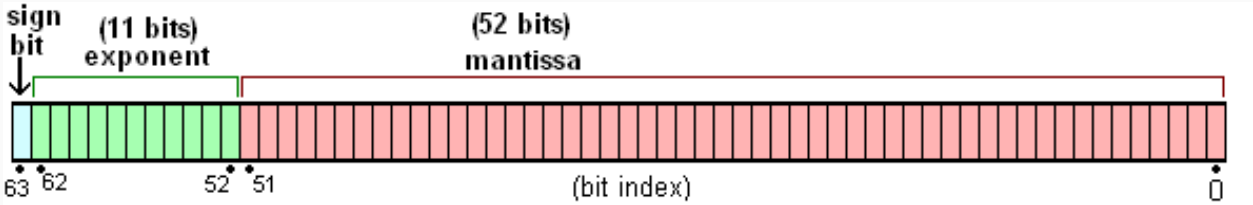
$$\underbrace{1}_{1}\,\underbrace{10\,000\,100}_{8}\,\underbrace{101\,000\,010\,000\,\ldots\,0}_{23}$$

ieee single precision

Special Cases:

- denormalized/subnormal numbers: use 1 extra bit in the significant: exponent is now -126 (less precision, more range), indicated by 00000000_2 in the exponent field
- two zeros: $+0$ and -0 (0 mantissa, 0 exponent)
- two ∞'s: $+\infty$ and $-\infty$
- ∞ (0 mantissa, 11111111_2 exponenet)
- NaN (any mantissa, 11111111_2 exponent)
- see appendix C.1 in NMC 6th ed.

ieee double precision



- 1-bit sign
- 11-bit exponent
- 52-bit mantissa
- single-precision: about 6 decimal digits of precision
- double-precision: about 15 decimal digits of precision
- $m = c - 1023$

precision vs. range

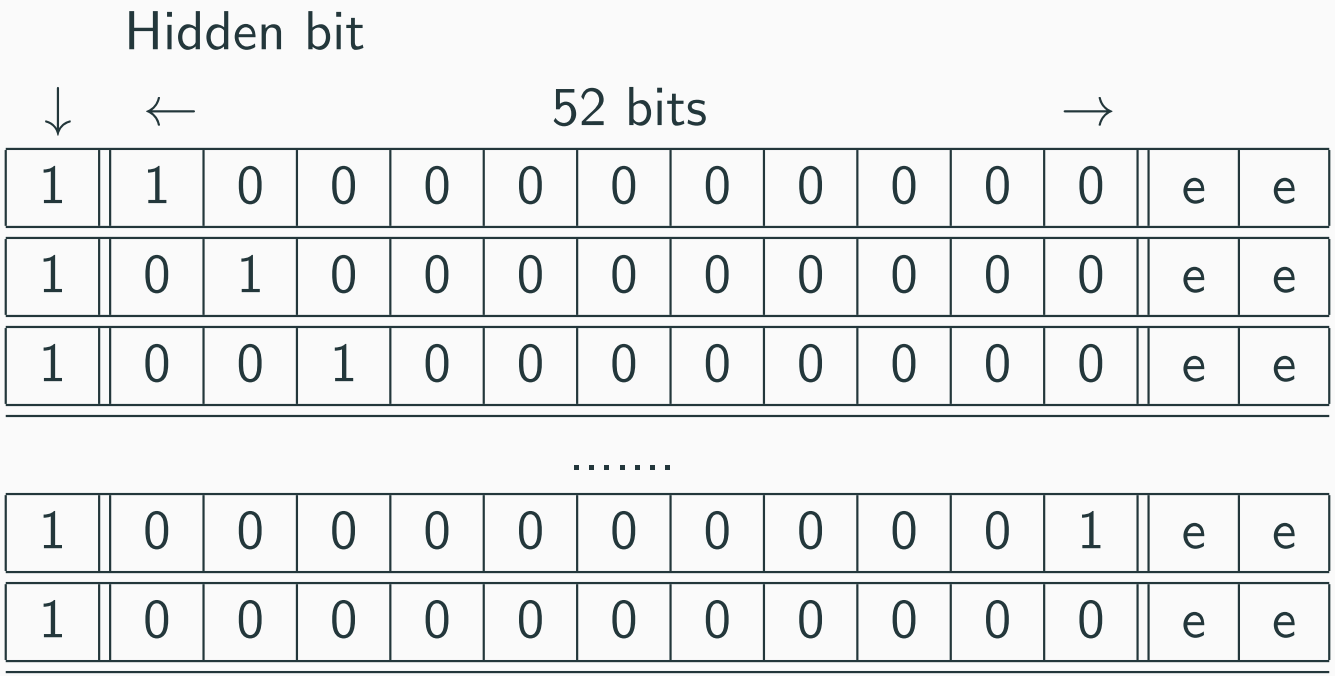
type	range	approx range
single	$-3.40 \times 10^{38} \leq x \leq -1.18 \times 10^{-38}$ 0 $1.18 \times 10^{-38} \leq x \leq 3.40 \times 10^{38}$	$2^{-126} \rightarrow 2^{128}$
double	$-1.80 \times 10^{318} \leq x \leq -2.23 \times 10^{-308}$ 0 $2.23 \times 10^{-308} \leq x \leq 1.80 \times 10^{308}$	$2^{-1022} \rightarrow 2^{1024}$

small numbers example

plus one example

ϵ_m

Take $x = 1.0$ and add $1/2, 1/4, \ldots, 2^{-i}$:



- Ooops!
- use $fl(x)$ to represent the floating point machine number for the real number x
- $fl(1 + 2^{-52}) \neq 1$, but $fl(1 + 2^{-53}) = 1$

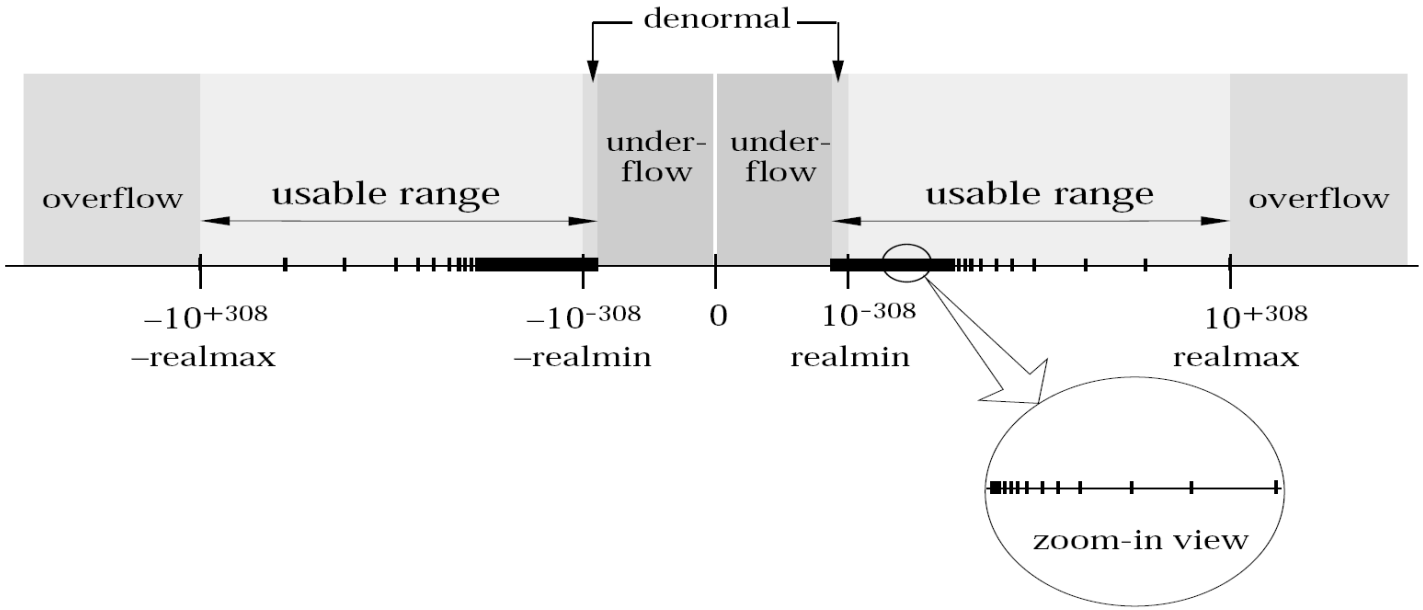
ϵ_m: machine epsilon

Machine epsilon ϵ_m is the smallest number such that

$$fl(1 + \epsilon_m) \neq 1$$

- The double precision machine epsilon is about 2^{-52}.
- The single precision machine epsilon is about 2^{-23}.

Floating Point Number Line



floating point errors

- Not all reals can be exactly represented as a machine floating point number. Then what?
- Round-off error
- IEEE options:
  - Round to next nearest FP (preferred), Round to 0, Round up, and Round down

Let x_+ and x_- be the two floating point machine numbers closest to x

- round to nearest: $round(x) = x_-$ or x_+, whichever is closest
- round toward 0: $round(x) = x_-$ or x_+, whichever is between 0 and x
- round toward $-\infty$ (down): $round(x) = x_-$
- round toward $+\infty$ (up): $round(x) = x_+$

floating point errors

How big is this error? Suppose (x is closer to x_-)

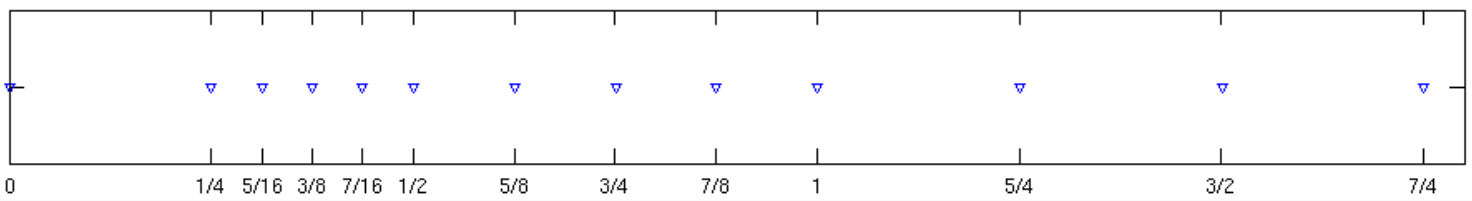
$$
\begin{aligned}
x &= (0.1 b_2 b_3 \dots b_{24} b_{25} b_{26})_2 \times 2^m \\
x_- &= (0.1 b_2 b_3 \dots b_{24})_2 \times 2^m \\
x_+ &= \left((0.1 b_2 b_3 \dots b_{24})_2 + 2^{-24}\right) \times 2^m \\
|x - x_-| &\leq \frac{|x_+ - x_-|}{2} = 2^{m-25} \\
\left|\frac{x - x_-}{x}\right| &\leq \frac{2^{m-25}}{1/2 \times 2^m} \leq 2^{-24} = \epsilon_m/2
\end{aligned}
$$

floating point arithmetic

- Problem: The set of representable machine numbers is FINITE.
- So not all math operations are well defined!
- Basic algebra breaks down in floating point arithmetic

Example

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



floating point arithmetic

Rule 1.

$$fl(x) = x(1+\epsilon), \quad \text{where} \quad |\epsilon| \leq \epsilon_m$$

Rule 2.

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

$$fl(x \odot y) = (x \odot y)(1+\epsilon_{\odot}), \quad \text{where} \quad |\epsilon_{\odot}| \leq \epsilon_m$$

Rule 3.

For $+, *$ operations

$$fl(a \odot b) = fl(b \odot a)$$

There were 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.

floating point arithmetic

Consider the sum of 3 numbers: $y = a + b + c$.

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

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

So disregarding the high order term $\epsilon_1\epsilon_2$

$$fl(fl(a+b)+c) = (a+b+c)(1+\epsilon_3) \quad \text{with} \quad \epsilon_3 \approx \frac{a+b}{a+b+c}\epsilon_1 + \epsilon_2$$

floating point arithmetic

If we redid the computation as $y_2 = fl(a + fl(b + c))$ we would find

$$fl(a + fl(b + c)) = (a + b + c)(1 + \epsilon_4) \quad \text{with} \quad \epsilon_4 \approx \frac{b + c}{a + b + c}\epsilon_1 + \epsilon_2$$

Main conclusion:

The first 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 more accurately, it is better to start with the small numbers first. [However, sorting before adding is usually not worth the cost!]

floating point arithmetic

One of the most serious problems in floating point arithmetic is that of cancellation. If two large and close-by numbers are subtracted the result (a small number) carries very few accurate digits (why?). This is fine if the result is not reused. If the result is part of another calculation, then there may be a serious problem

Example

Roots of the equation

$$x^2 + 2px - q = 0$$

Assume we want the root with smallest absolute value:

$$y = -p + \sqrt{p^2 + q} = \frac{q}{p + \sqrt{p^2 + q}}$$

catastrophic cancellation

Adding $c = a + b$ will result in a large error if

- $a \gg b$
- $a \ll b$

Let

$$a = x.xxx \cdots \times 10^{0}$$
$$b = y.yyy \cdots \times 10^{-8}$$

Then

$$\begin{array}{cc} & \overbrace{x.xxx\ xxxx\ xxxx\ xxxx}^{\text{finite precision}} \\ + & 0.000\,0000\ yyyy\ yyyy \quad yyyy\ yyyy \\ \hline = & x.xxx\ xxxx\ zzzz\ zzzz \quad \underbrace{????\ ????}_{\text{lost precision}} \end{array}$$

catastrophic cancellation

Subtracting $c = a - b$ will result in large error if $a \approx b$. For example

$$a = x.xxxx\ xxxx\ xxx1\ \overbrace{ssss\ldots}^{\text{lost}}$$

$$b = x.xxxx\ xxxx\ xxx0\ \overbrace{tttt\ldots}^{\text{lost}}$$

Then

$$\begin{array}{cc} & \overbrace{x.xxx\ xxxx\ xxx1}^{\text{finite precision}} \\ + & x.xxx\ xxxx\ xxx0 \\ \hline = & 0.000\,0000\,0001 \quad \underbrace{????\ ????}_{\text{lost precision}} \end{array}$$

summary

- addition: $c = a + b$ if $a \gg b$ or $a \ll b$
- subtraction: $c = a - b$ if $a \approx b$
- catastrophic: caused by a single operation, not by an accumulation of errors
- can often be fixed by mathematical rearrangement

loss of significance

Example

$x = 0.37214\,48693$ and $y = 0.37202\,14371$. What is the relative error in $x - y$ in a computer with 5 decimal digits of accuracy?

$$\frac{|x - y - (\bar{x} - \bar{y})|}{|x - y|} = \frac{|0.37214\,48693 - 0.37202\,14371 - 0.37214 + 0.37202|}{|0.37214\,48693 - 0.37202\,14371|}$$
$$\approx 3 \times 10^{-2}$$

loss of significance

Loss of Precision Theorem

Let x and y be (normalized) floating point machine numbers with $x > y > 0$.

If $2^{-p} \leq 1 - \frac{y}{x} \leq 2^{-q}$ for positive integers p and q, the significant binary digits lost in calculating $x - y$ is between q and p.

loss of significance

Example

Consider $x = 37.593621$ and $y = 37.584216$.

$$2^{-11} < 1 - \frac{y}{x} = 0.00025\,01754 < 2^{-12}$$

So we lose 11 or 12 bits in the computation of $x - y$. yikes!

Example

Back to the other example (5 digits): $x = 0.37214$ and $y = 0.37202$.

$$10^{-4} < 1 - \frac{y}{x} = 0.00032 < 10^{-5}$$

So we lose 4 or 5 bits in the computation of $x - y$. Here, $x - y = 0.00012$

which has only 1 significant digit that we can be sure about

loss of significance

So what to do? Mainly rearrangement.

$$f(x) = \sqrt{x^2 + 1} - 1$$

loss of significance

So what to do? Mainly rearrangement.

$$f(x) = \sqrt{x^2 + 1} - 1$$

Problem at $x \approx 0$.

loss of significance

So what to do? Mainly rearrangement.

$$f(x) = \sqrt{x^2 + 1} - 1$$

Problem at $x \approx 0$.

One type of fix:

$$f(x) = \left(\sqrt{x^2 + 1} - 1\right)\left(\frac{\sqrt{x^2 + 1} + 1}{\sqrt{x^2 + 1} + 1}\right)$$

$$= \frac{x^2}{\sqrt{x^2 + 1} + 1}$$

no subtraction!