你所不知道的 C 語言: 浮點數運算

直播錄影

點題

呂紹榕 （Louie Lu） 說

> 「寫程式三大錯覺：我懂**浮點數**、我懂 **Unicode**、我懂**時區**」

Precision (精密度) 和 Accuracy (準確度)

- 倘若我們的數值系統只能保證十進位小數點後 10 位得以正確表達
- 寫成 pi = 3.1415926535 就是 Precise + Accurate
- 寫作 pi = 3.141 則是 3.141 是 Imprecise + Accurate
- 寫作 pi = 3.1415831786 是 Precise + Inaccurate
- 寫作 pi = 3.275 就是 Imprecise + Inaccurate

工程領域往往是一系列的取捨結果，浮點數更是如此，在軟體開發有太多失誤案例源自工程人員對浮點數運算的掌握不足，本議程希望藉由探討真實世界的血淋淋案例，帶著學員思考 IEEE 754 規格和相關軟硬體考量點，除了關注浮點數原理，我們也會用 C 程式來操作浮點數內部，並談及定點數運算。最後也會探討在深度學習領域為了改善資料處理效率，而引入的 BFloat16 這樣的新標準。

在許多演算來說由於同時對於 precision 與 dynamic range 的需求, 因此在計算過程中對於浮點數的使用很常見 (若要避免使用會有很高的專業與困難度)，浮點數運算主要優點在於可表示極大與極小值，相較整數能大幅避免 overflow 與 underflow，缺點是有效位數的減少，而且現今多數的計算單元都俱備浮點數運算的支援 (Intel 486 與之前的時代，FPU 是高檔貨，ARM 自 ARMv7-A 才列入標準配置)。然而若橫跨個人電腦、手機、CPU 與 GPU, CPU 與 DSP, 甚至於結合其中若干者，浮點數就變成非常難以考量與處理的負擔, 而為了區分是程式錯誤或誤差就必須耗費相當的心力。

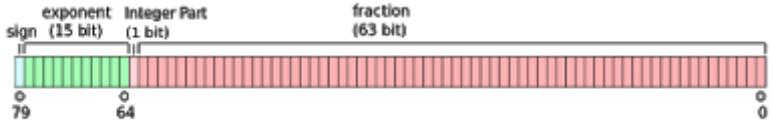
IEEE 754 定義的不僅僅只是 format 而已, 還有著 rounding mode, required operations 以及 exception handling，符合 IEEE 754 的規範下，才可能有相同的輸出結果 (這當然只是最低門檻)。

Format 本身的問題

扣除浮點數因格式問題不可能表示全部的數目外，格式本身最大的問題是因為 dynamic range 的移動, 像是 `(A + B) + C` 與 `A + (B + C)` 單以代數考量這無疑是相等的, 但是若以浮點數格式去思考, 你就會意會到輸出結果很有可能會不同, 而原始演算實作所採用的累加或相乘的順序, 必須在優化實作上努力維持才能產生一致的輸出結果, 這樣的問題對於程式優化影響最大的部分是平行計算, 無論 TLP or ILP, 因為平行度優化考量, 都會有分割與不同面向個別累計的需求, 如此勢必都會產生一定的誤差

CPU 間的問題

或許有些人會認為只使用 CPU 那麼就不會碰到浮點數問題了, 這樣說只算對了一半, 而且你還是必須要只使用一種平台以及指令集, 對於多數演算法設計工程師而言, 他們很慣於使用個人電腦平台, 甚至會使用 MATLAB，x86 PC 上的程式預設會使用 x87 浮點數協同單元 指令集, 而這是許多問題的開始 - x87 內部使用 80bit 浮點數表示



通常 x86 CPU 是在 x87 FPU 的內部以 80 bits 計算得到結果後, 再 truncate 為 IEEE-754 的 float(32b) 或 double(64b), 這就表示這與 IEEE 754 FPU 的輸出結果會有微小的差異, 目前常見的手機的 ARM 架構, 即為 IEEE 754 compatible FPU, 所以光是 ARM 與 x86 PC 相同的程式碼其輸出結果基本上就會有所不同, 而對於 ARM 與 x86 的部分, 就必須仰賴以 IEEE 754 設計的 SSE2 指令集

- gcc 與 clang 很早即可藉由 `-mfpmath=sse2` 編譯參數來達到，但 Visual Studio 必須是 2013 版後才有正確的機械碼輸出

> 相關套件: sse2neon

對於 ARM 與 x86 平台的一致性方案反而又揭露了另一個層面問題:

> 就算使用了單一CPU架構, 在 ISA 指令集間的支援還是會有所不一致!

類似於 x86 平台上 x87 指令與 SSE2 指令有著不同輸出, 同樣地 Arm VFP 指令 (IEEE 754 相容) 與 NEON 指令(非完全 IEEE 754 相容) 也可能會有輸出結果不同, 而這樣的問題還會再帶到 libmath 的實作方式, 讓要處理一致性的問題再度的變得更嚴重

GPU

GPU 本身有著龐大的浮點數計算能力, 但是通常為了能達到更高的吞吐量以及加速上的考量, 在計算結果與 IEEE-754 可能存在差異, 不同代的 GPU 或是不同架構都有可能有所不同. 像是 CUDA 是在 compute compatibilty v2.0 之後才完備了 IEEE 754 的支援, 除此之外許多硬體加速的數學函數的輸出上也不保證與 CPU 一致, 這點 NVIDIA 在 2011 GTC 中給的 Floating Point and IEEE-754 Compliance for Nvidia GPUs 簡報中有很詳盡的說明. 對於其他 GPU 以及各 CPU/GPU 平台上的 OpenCL 中的 built-in functions 的實作與支援也有著相同的道理.

DSP

對於 DSP 而言這樣的痛苦並不在於 IEEE 754 本身, 而是多數的多媒體面向的 DSP 為了考量計算能力與面積, 結果多半是直接不俱備浮點數運算能力, 像是 Qualcomm S82x 中的 Hexagon 680 HVX 就不俱備 floating 運算的 SIMD 指令, 而通常的處理作法是採用 fixed-point (quantization) 的浮點模擬, 然而若採用靜態位數的方式容易失真, 而動態的方式有著實作上的複雜度以及多餘計算的負擔. 而數學函數上的實作若難以避免則通常必須透過相當迂迴的方式.

Lookup-Table or Frame-based Parameters

對於跨裝置的正確性驗證, 由單一裝置輸出的 Lookup Table(單一的 math function 像是 sin, cos, log, exp 等等) 或是一整張預先透過單一裝置計算的 Frame-based Parameters(複雜的並結合多個 math function 的運算) , 是常用來確認誤差單純是由 floating 計算造成的技巧. 以此來確保實作上的流程與邏輯無誤.

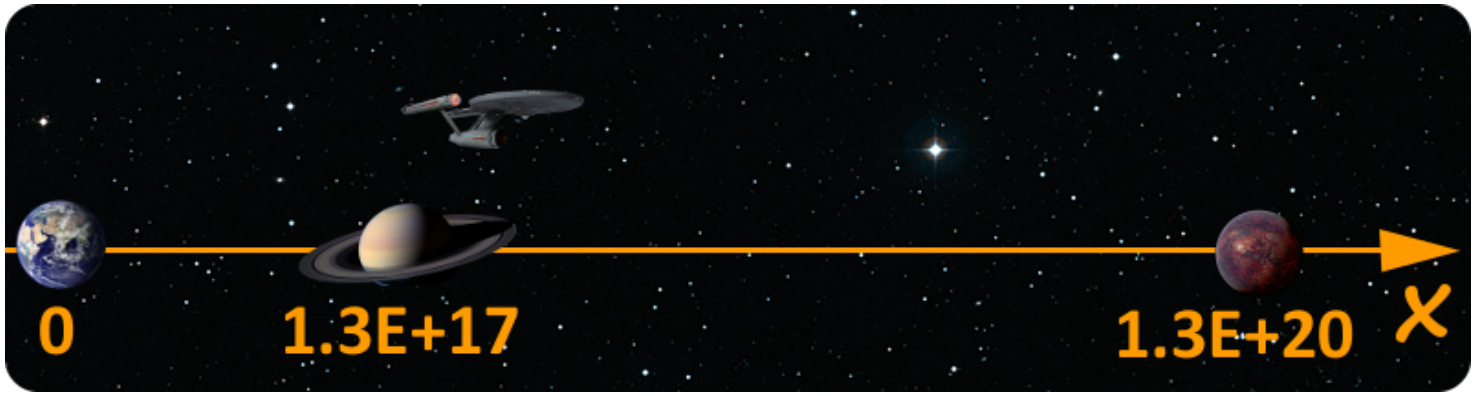
延伸閱讀: The pitfalls of verifying floating-point computations

浮點數之旅

導讀

- 從根號 2 的運算談浮點數

Floating point precision is not uniformly distributed



- The further into the space, the bigger the interval between the two adjacent values
- At some point the added value is less than this interval, and the ship stops

Binary representation of the floating point numbers

```
$ python
>>> 0.1 + 0.2 == 0.3
False
>>> 0.1 + 0.2
0.30000000000000004
>>> 0.3
0.29999999999999999
```

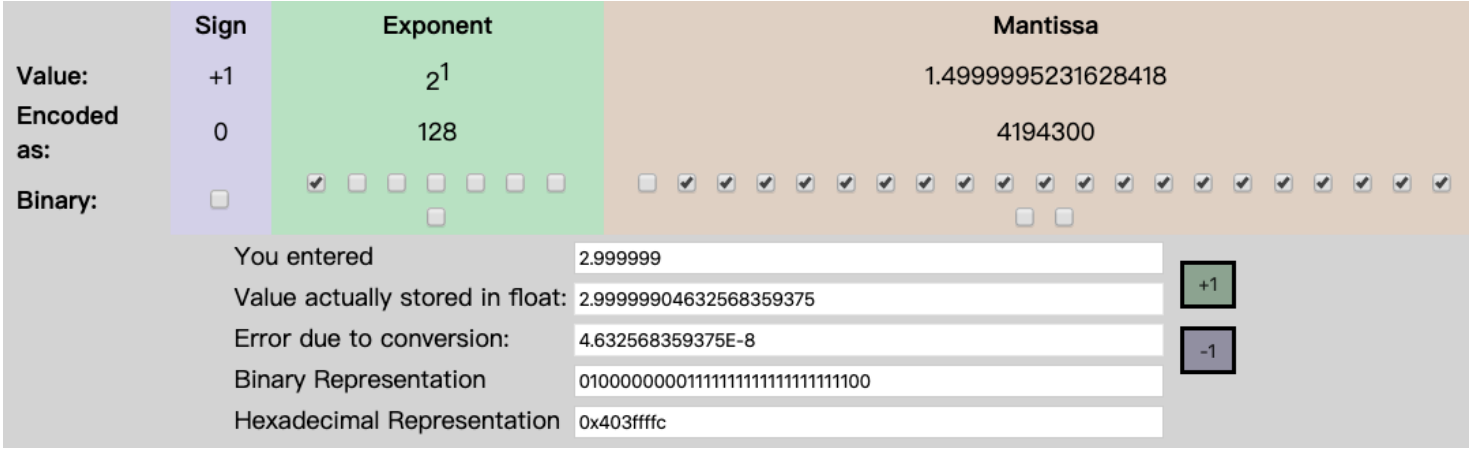
- 0.1 is not “0.1”
- It’s “0.0001100110011001(1001)…”
- None of these numbers (0.1, 0.2, 0.3) are represented exactly
- Humans have ten fingers, computers have two
- Almost all of the computing hardware nowadays

```c
#include <stdio.h>
#include <math.h>

int main(){
    float f1 = 2.999999;
    float f2 = 2.9999999;
    printf("floor(f1)=%10.20f\nfloor(f2)=%10.20f\n",
                    floor(f1), floor(f2));
    return 0;
}
```

執行結果:

```
floor(f1)=2.00000000000000000000
floor(f2)=3.00000000000000000000
```

使用 IEEE-754 Floating Point Converter:



當對單精度的 f1 存入 `2.999999` 時，其實際存起來的的值是 `2.99999904632568359375`。有效位為小數後六位（最低位 2^{-23} = 0.000000119209... 為可存的離散數間的最小間隔）。對這個值取

floor，自然就是得到 2.0。當對單精度 f2 存入 `2.9999999` 時，實際存起來的值是 `3.0`，對 3.0 取 floor 會得到 3.0。

Anecdote! In Soviet Russia it's ternary

- Built in 1958 at the Moscow State University
- Balanced ternary, usesing the three digits: -1, 0, and +1
- Natural representation of negative numbers (no "sign bit")
- Some computations are more efficient
- Donald Knuth argued it might be used in future hardware
- …actually not that bad idea!

延伸閱讀:

- 你所不知道的 C 語言：數值系統篇

Decimal to binary conversion: whole numbers

- Always finite representation
- Iteratively divide by 2, modulo goes into binary digits, to the left

Current Decimal	Current Binary
135	
67	1
33	11
16	111
8	0111
4	00111
2	000111
1	0000111
	10000111

- 135_{10} == 10000111_{2}

Decimal to binary conversion: fractions

- Iteratively multiply by 2, leftmost digit goes into the binary digits, to the right

Current Decimal	Current Binary
0.6875	
[1]0.375	1
[0]0.75	10
[1]0.5	101
[1]	1011

- 0.6875_{10} == 0.1011_{2}

Decimal to binary conversion: fractions

- However, only finite when power of two in denominator
- 0.6875 = $\frac{11}{16}$, 16 = 2^4

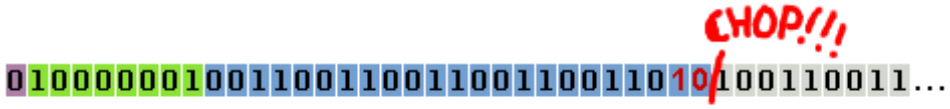
Current Decimal	Current Binary
0.8	
[1]0.6	1
[1]0.2	11
[0]0.4	110
[0]0.8	1100
[1]0.6	11001

Current Decimal	Current Binary
…	…

- Goes into infinite loop. 0.8_{10} == $0.11001100(1100)\ldots_2$

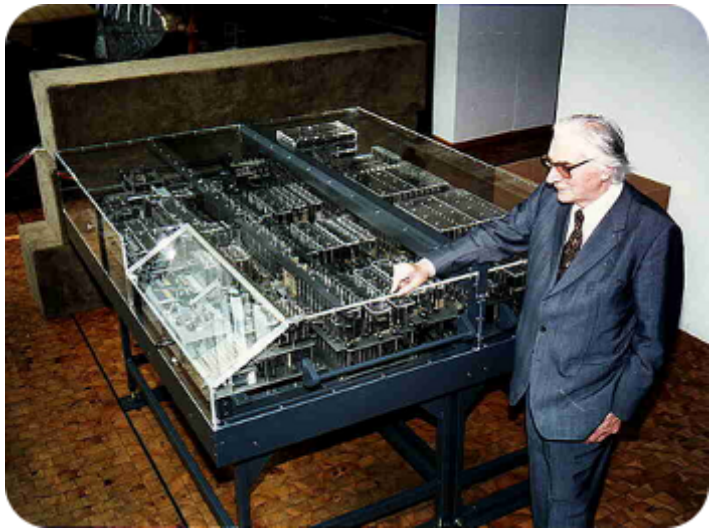
Inexact representation: rounding

- What to do with these infinite fractions?..
- Chop, of course!



- The last two digits get modified because of rounding
- The image shows the standard way of representing it, in “single precision”

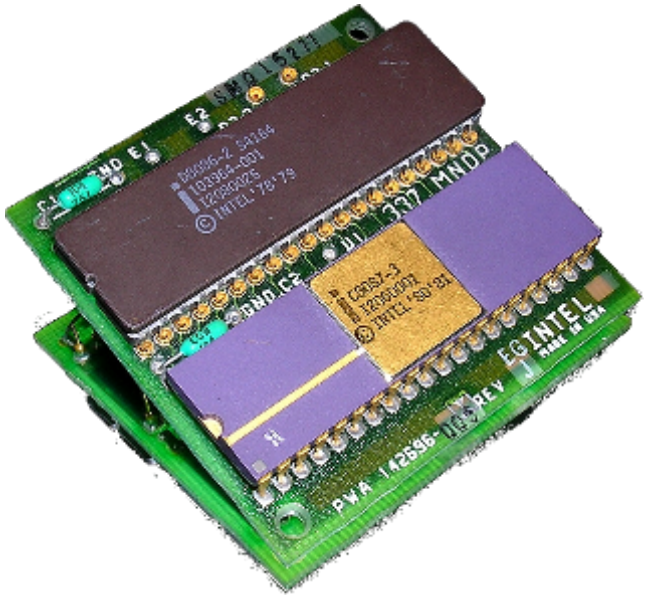
The first Floating-Point Hardware Ever



- Z1, built in 1935-1938 by Konrad Zuse
- The first programmable machine, supported floating-point computations
- Had all the basic ingredients of modern machines
- Binary system
- 22 bits with 14 mantissa bits (with one extra implied), a 7-bit exponent, and a sign bit
- Z2, Z3, Z4 (32bit)

可程式化電子計算機 ENIAC 在 1946 年的 2 月誕生於美國賓夕法尼亞大學，當時它的計算能力是每秒 5000 次加法或 400 次乘法。ENIAC 的使用者是美國陸軍的彈道研究實驗室，最初打算是用於計算火炮的火力表（即彈丸的彈道）。可是後來 John von Neumann 參與到專案裡面來，他當時正在研究氫彈，所以 ENIAC 的第一次測試運行是計算氫彈的相關數據。

Intel 8087: First Floating Point Coprocessor



- Announced in 1980
- A separate microprocessor, working together with 8086
- Speed up floating point computations up to 500%
- Uses stack-based register model, unlike main CPU
- 80-bit internal register precision
- Later CPUs include the functionality in the main CPU
- Triggered the IEEE 754 standard

Anecdote! Pentium FDIV bug

- Hardware problem with Floating Point Division
- Certain Intel P5 CPU models, introduced in 1993
- A problem discovered by professor Thomas Nicely
- Rare: 1 in 9 billion floating point divides
- Could be verified via division $\frac{4195835}{3145727}$:
  - `1.333739068902037589` instead of: `1.333820449136241002`
- …public outcry, Intel offered replacement

延伸閱讀:

- 軟體缺失導致的危害

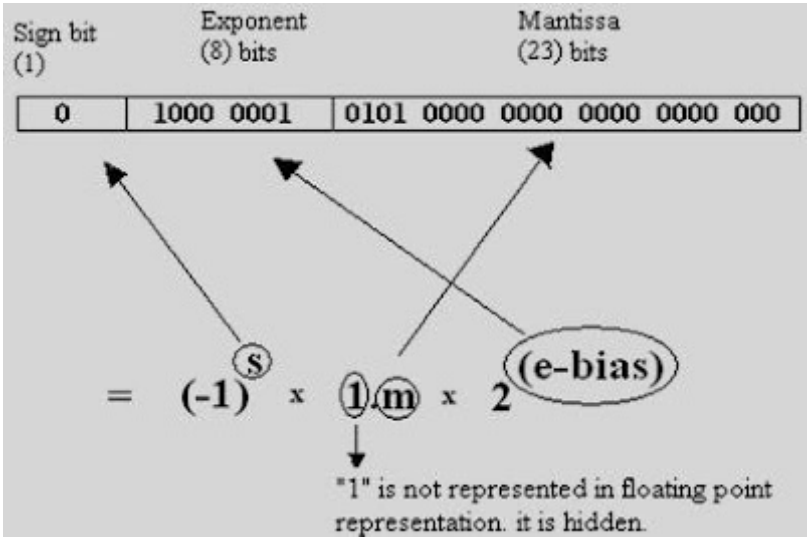
IEEE 754 Standard

> William Kahan 教授

微處理器出現前，很多大型電腦具備浮點運算的能力，不過各自有其浮點表示方式；1976 年，Intel 發表 8086 後，著手設計一款可跟 8086/88 搭配的浮點運算器，這個計劃的主持人 John Palmer 提議將這顆浮點運算器的數值格式和運算規則公開成標準，於是 Intel 聘請加州大學柏克萊分校的 William Kahan 教授為顧問，向 IEEE 提交了第一個浮點運算標準，亦即現在的 IEEE-754。William Kahan 教授也因對於浮點運算標準化的貢獻，獲得 1989 年的 Turing Award。

> An Interview with the Old Man of Floating-Point

一開始 IEEE-754 制定時，主要成員來自微處理器領域的新秀，不僅有 Intel，當時 Zilog, Motorola 也沒缺席，Digital Equipment Corporation (簡稱 DEC) 也相當活躍，不過當時大型和超級電腦的巨頭，如 Cray 和 CDC，因為看不上前述廠商和其市場，所以缺席；雖然草案由 Intel 提交，但 DEC 不是省油的燈，根據後者既有的 VAX 電腦的浮點運算機制，制定出另一套方案，於是爭論自 1977 年持續到 1981 年，主要爭論焦點是 Denormal number。

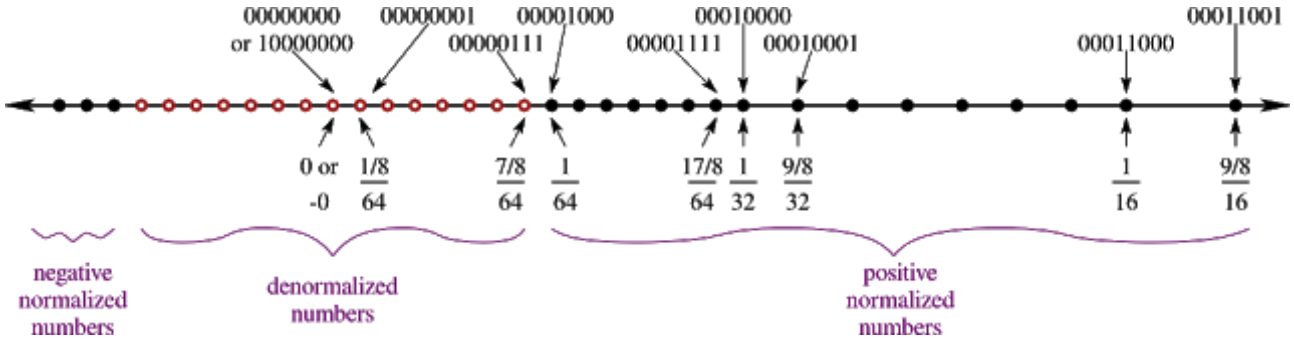


一個「正規」(normal) 浮點數基本上分成三個部份:

- 1 bit 的正負號
- 指數
- 尾數

指數用來表示數值範圍，尾數則是數值的精確度，若這兩個欄位皆為 `0`，就是我們在數學意義上的 `0` (有正負號)，指數部份若全都是 `1`，表示是無限大或者 NaN，其他狀況下，指數基本上最小就是 1 (即 `0x01`)，這衍生的問題是，有時計算時會出現這種情況「指數是 `0`，後面尾數卻不是 `0`」，稱為 underflow，意思是運算出來的數值非常接近 `0`，近到無法用正規的浮點數來表示，怎麼處理呢？DEC 的方法是直接當作 `0`，Intel 則是提出一個複雜的方案：把這類數值特別處理，使其可表示比正規浮數字更接近 `0` 的範圍，這樣的數字就稱為 Denormal number

> underflow 會有什麼麻煩呢？明明 $A > B$，但是 $A - B$ 的運算結果卻成為 `0`

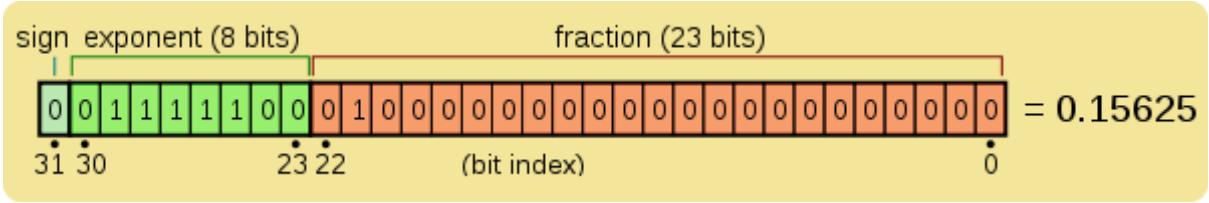


DEC 採用的方法稱為 abrupt underflow (突然式下溢位)，而 Intel 的提案稱為 gradual underflow (漸進式下溢位)，當時許多電腦採用跟 DEC 相同的方法，畢竟實作和理解都容易些，但是 Intel 認定該方式會讓 `0` 到最小非 `0` 浮點數之間的差距過大。最後在 1981年，DEC 聘請一位馬里蘭大學的錯誤分析專家 G.W. (Pete) Stewart III 教授替他們的方案辯護，但仔細研究後，Stewart 教授認為 gradual underflow 是較好的方案，勸 DEC 放棄爭論。1985 年，IEEE-754-1985 正式通過，成為浮點運算標準，儘管 Intel 在 1980 年發表 8087、1983 年發表搭配 80286 的 80287 協同處理器，這兩款浮點運算器尚未完全符合 IEEE-754 規範，直到 1987 年的 80387 才是首款真正符合 IEEE-754-1985 標準的浮點運算器。

IEEE 754 Standard: Scope

- Basic and extended floating-point number formats
- Add, subtract, multiply, divide, square root, remainder, and compare operations
- Conversions between integer and floating-point
- Conversion from/to decimal string
- Exceptions and their handling

Single precision: binary32



- The actual value on the picture:

> $0.15625_{10} = 0.00101_2 = 1.01 \times 2^{-3}$

- s: sign, 1 bit
- e: exponent, 8 bits
- Exponent is “biased” by 127 (1111100 = 124, 124-127 = -3)
- m: mantissa, 23 bits
- One bit in mantissa is implicit, “always present”
- This is called “normalized” representation

$$v = (-1)^s(1 + \sum_{i=1}^{23} b_{23-i}2^{-i})2^{e-127}$$

“Seven digits after the dot”

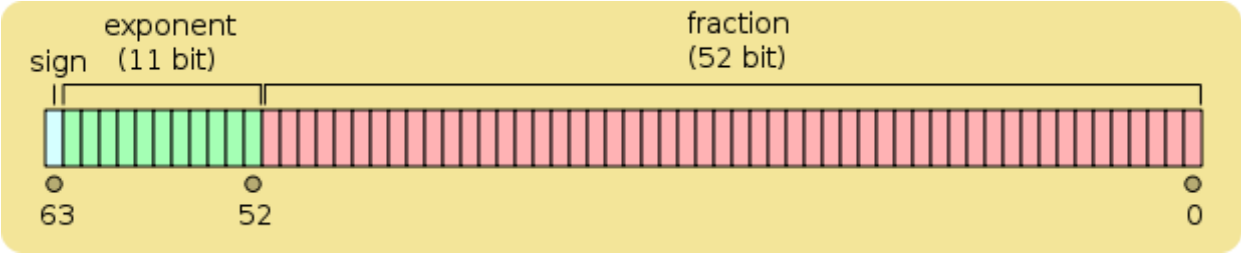
- “The amount of precision that a floating number can have”
- Urban myth… but partially true:

> $\log_{10} 2^{24} \approx 7.224719$

- 24 because: 23 bits for mantissa + 1 implicit bit
- In C++, std::numeric_limits::digits10 (in C, macro FLT_DIG) gives 6.
- Wikipedia says “6 to 9”. Get the same number if converting Decimal $\rightarrow$ IEEE 754 $\rightarrow$ Decimal

- All decimals with 6 digits survive “trip and back”
- But technically, from 0 to 100+ digits: denormals have fewer, exact numbers have more

Double precision: binary64

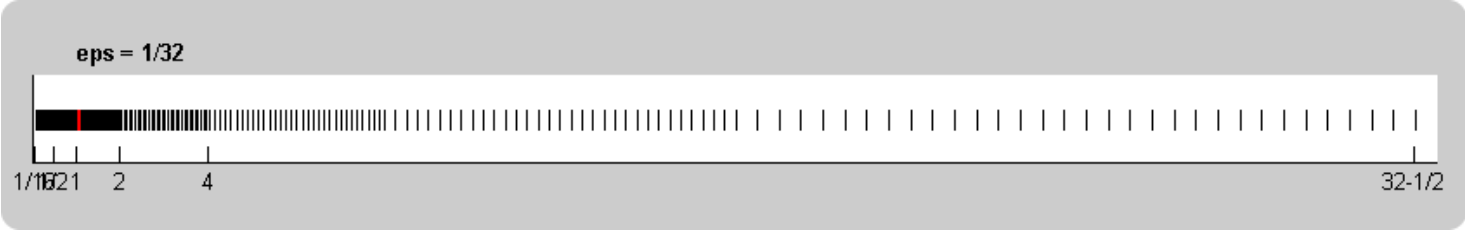


- s: sign, 1 bit
- e: exponent, 11 bits
- m: mantissa, 52 bits
- Same as binary32, but uses twice the amount of bits for everything
- Considerably better precision

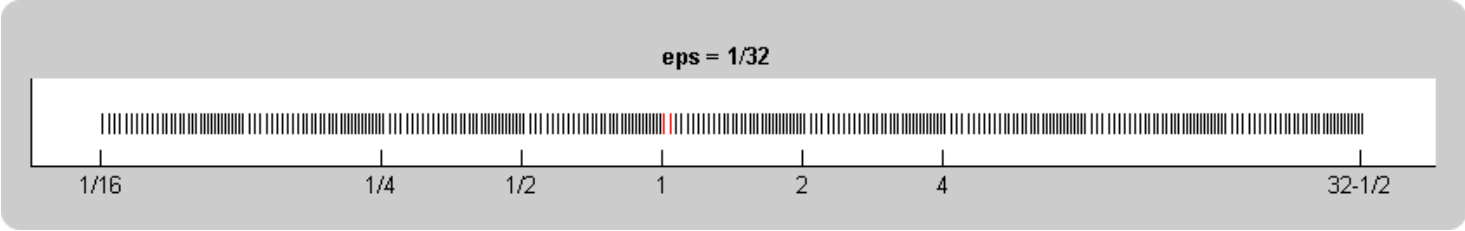
$$v = (-1)^s(1 + \sum_{i=1}^{52} b_{52-i}2^{-i})2^{e-1023}$$

Uneven density of the floating point numbers

- Precision decreases with increasing magnitude

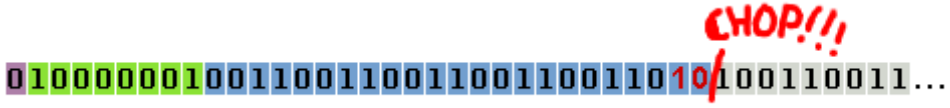


- Because of exponent/mantissa representation
- That's why it is called "floating point"
- Actually, logarithmic:



Rounding

- If digits do not fit, they get chopped (rounded):



- The last two digits get modified because of rounding
- There are different ways of rounding. IEEE 754 suggests 5
- The default rounding mode is "round to nearest, tie to even"

Rounding modes

Rounding mode	+11.5	+12.5	-11.5	-12.5
to nearest, ties to even	+12.0	+12.0	-12.0	-12.0
to nearest, ties from zero	+12.0	+13.0	-12.0	-13.0

Rounding mode	+11.5	+12.5	-11.5	-12.5
toward 0	+11.0	+12.0	-11.0	-12.0
toward +inf	+12.0	+13.0	-11.0	-12.0
toward -inf	+11.0	+12.0	-12.0	-13.0

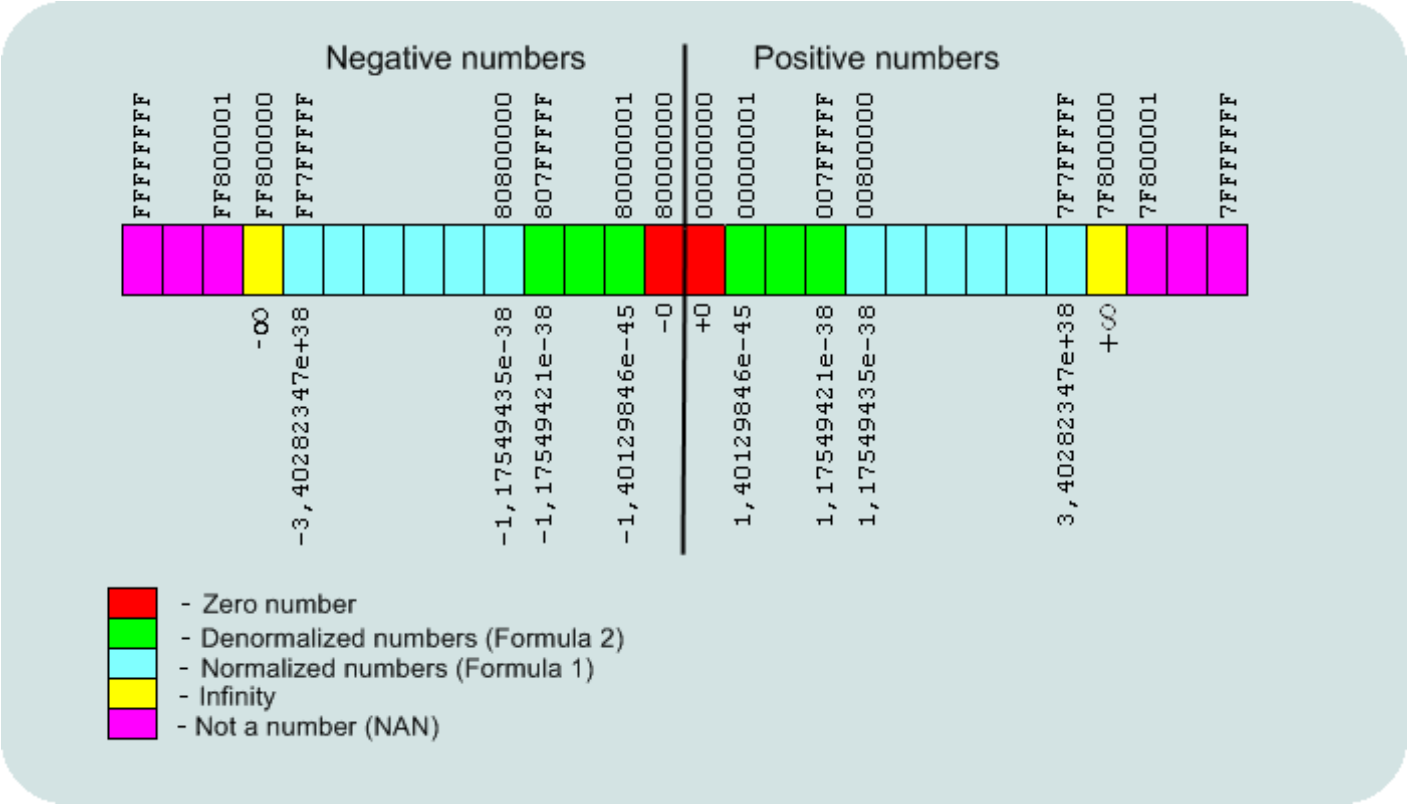
Subnormal numbers

- “Gradual underflow”, AKA “denormals”

> underflow 指浮點數計算的結果小於可以表示的最小數。underflow 出現在計算結果很接近零，使得計算結果小於浮點數可以表示的最小數字。算術下溢也可以視為是浮點數指數在負值時的溢位。例如，浮點數指數範圍為 -128 至 127，一個絕對值小於 2^{-127} 的浮點數就會造成下溢（假設 -128 用於表示負無窮

- Very useful for some algorithms (e.g. prevents divizion by 0)
- But generally very slow! (because of special treatment) (take an average time of 15.63 ns on a i7-8700 cpu)
- No implicit leading 1! (i.e. binary not normalized)
- Use less bits for mantissa. Different representation
- For C++/FORTRAN, can be disabled on compiler level (`flush-to-zero` and `denormals-are-zero`)

Special values



- `+0` and `-0` (they are equal by the standard!)
- `+Inf` and `-Inf`
- `NaN` (not-a-number), `QNaN` (quiet not-a-number)

	1	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**X**		**...**

1. if the ten-th bit is 1, then it is QNaN
2. ten-th bit is 0, then it is NaN (signaling NaN)
3. In order to produce a NaN manually, this will only tranform the 0x7F800001 into a int-like value

```
float a = (float) 0x7F800001
```

4. You have to cast it with pointer

```
*(int *) &f2 = 0x7F800001;
```

5. When a floating point operation produce quiet NaN, it will not interrupt the whole operation, the programmer have to check the value by their own. According to IEEE 754 7.2 section, qNaN are produced when

- any general-computational or signaling-computational operation on a signaling NaN (see 6.2), except for some conversions (see 5.12)
  $\rightarrow$ operation involved in NaN (just like NaN will contaminate the whole operation)
- multiplication: multiplication(0, ∞) or multiplication(∞, 0) fusedMultiplyAdd: fusedMultiplyAdd(0, ∞, c) or fusedMultiplyAdd(∞, 0, c) unless c is a quiet NaN; if c is a quiet NaN then it is implementation defined whether the invalid operation exception is signaled
  $\rightarrow$ multiply 0 with ∞
- addition or subtraction or fusedMultiplyAdd: magnitude subtraction of infinities, such as: addition(+∞, −∞)
  $\rightarrow$ add/subtract ∞ with ∞
- division: division(0, 0) or division(∞, ∞)
  $\rightarrow$ divid ∞/0 with ∞/0
- remainder: remainder(x, y), when y is zero or x is infinite and neither is NaN
- squareRoot if the operand is less than zero
- quantize when the result does not fit in the destination format or when one operand is finite and the other is infinite

6. operation that produce signaling NaN will interrupt the operation, and send out a signal, which includes
    - conversion of a floating-point number to an integer format, when the source is NaN, infinity, or a value that would convert to an integer outside the range of the result format under the applicable rounding attribute
    - comparison by way of unordered-signaling predicates listed in Table 5.2, when the operands are unordered
    - logB(NaN), logB(∞), or logB(0) when logBFormat is an integer format (see 5.3.3).
        - While IEEE 754 define most of the API outcome of INF's sign bit, according to chapter 6.3, there is little regulation on how NaN's sign bit should be

Exceptions

- 5 types of exceptions should be signaled when detected
- Invalid operation: involving NaN, sqrt(-1), float $\rightarrow$ int overflow
- Division by zero. The result is signed Inf.
- Overflow: when rounding and converting
- Underflow
- Inexact

Anecdote! USS Yorktown (CG-48)

- On 21 Sept. 1997, maneuvering off the coast of Cape Charles
- Crewman entered a blank fiend into database
- The blank was treated as zero
- Caused Divide-by-Zero Exception
- which was not handled properly by the software
- The exception was propagated to the system (WinNT 4.0)
- …the vessel paralyzed for 3 hours

Exceptions: Traps

- User should be able to specify a handler (or disable)
- Many programming languages just don’t care (such as Java)
- C/C++ has an API:

```
void main() {
    unsigned int fp_control_word, new_fp_control_word;

    _controlfp_s(&fp_control_word, 0, 0);
```

```
    // make the new fp env same as the old one,
    // except for the changes we're going to make
    new_fp_control_word = fp_control_word |
        _EM_INVALID | _EM_DENORMAL | _EM_ZERODIVIDE |
        _EM_OVERFLOW | _EM_UNDERFLOW | _EM_INEXACT;
    // update the control word with our changes
    _controlfp_s(&fp_control_word, new_fp_control_word, _MCW_EM)
}
```

Anecdote! Ariane5 rocket launch

- Launched in June 1996, 10 years and $7 billion to build
- Crashed in 36.7 seconds after launch
- 64 bit floating point number (horizontal velocity of the rocket) converted to 16 bit integer
- The number bigger than in Ariane4; unchecked overflow
- Internal software exception, shut down the navigation system
- …the ultimate cause is the lack of proper FP exception handling

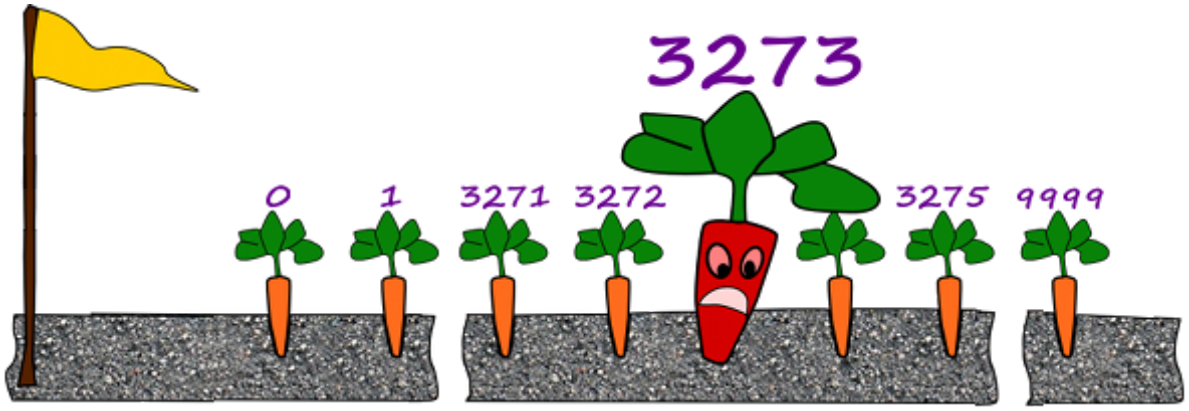
Carrots Planting

```
#define NUM_CARROTS 10000          /* the number of carrotts to plant */
const float BOBWALK_WIDTH  = 4.8f; /* the width of the bobwalk (feet) */
const float PLANT_INTERVAL = 1.25f;/* the planting interval (feet) */

float carrot_pos[NUM_CARROTS];     /* carrot positions go here */

int main() {
    int i;
    /* Plant the carrots! */
    for (i = 0; i < NUM_CARROTS; i++) {
        carrot_pos[i] = BOBWALK_WIDTH + i*PLANT_INTERVAL;
        printf("Carrot %d goes to %g feet!\n", i, carrot_pos[i]);
    }
    return 0;
}
```

```
const float EPSILON = 0.0001f; /* epsilon to check placement */
void test_carrots() {
    /* verify that carrots are all planted evenly! */
    int i, num_bad = 0;
    for (i = 1; i < NUM_CARROTS; i++) {
        float d = carrot_pos[i] - carrot_pos[i - 1];
        if (fabs(d - PLANT_INTERVAL) > EPSILON) {
            printf( "The carrot %d is planted wrong!!!\n" , i);
            num_bad++;
        }
    }
    if (num_bad == 0) printf( "All carrots placed well!!!");
    else printf("%d of %d carrots placed incorrectly... :(",
        num_bad, NUM_CARROTS);
}
```

Output:

```
The carrot 3273 is planted wrong!!!
1 of 10000 carrots placed incorrectly... :(
```




```
offs = single(4.8) + single(1.25)*[0:9999];
delta = offs(2:end) - offs(1:end-1);
error = delta - single(1.25);

plot(error, 'r-', 'LineWidth', 2);
grid on;
title('Planting distance error');
xlabel('Carrot index');
ylabel('Error (feet)');
```


```
octave> find(abs(error) >= 1e-4)
ans =  3273
octave> non_exact = find(abs(error) != 0)
non_exact =
      9     48    201    816   3273
octave> non_exact_offsets = 4.8 + non_exact*1.25
non_exact_offsets =
     16.050     64.800    256.050   1024.800   4096.050
octave> error(non_exact)
ans =
  -9.5367e-007  3.8147e-006  -1.5259e-005  6.1035e-005  -2.4414e-00
```

- In majority of the cases the rounding errors cancel each other
- But in a few curious cases (around values 16, 64, 256, 1024, 4096) there is a catastrophic cancellation
- Rounding error in the lowest bits was enough

Problem: Catastrophic cancellation

$c = 10^{10} * (a - b)$
a = 31.006276686241002
b = 31.006276685299816

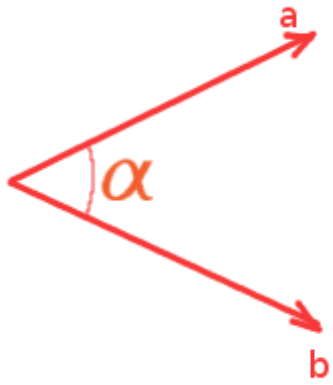
- a and b are close to each other
- a and b have some garbage in the lower bits
- After the substraction, the “good bits” cancel each other:

> 31.006276686241002 - 31.006276685299816 = 0.000000000941186

- The garbage gets promoted and claims to be a meaningful computation result:

> 10000000000 * 0.000000000941186 = 9.41186

Problem: Going out of the definition domain



$a \cdot b = \|a\| \, \|b\| \cos \alpha$

$$\alpha = \arccos \frac{a}{\|a\|} \cdot \frac{b}{\|b\|}$$

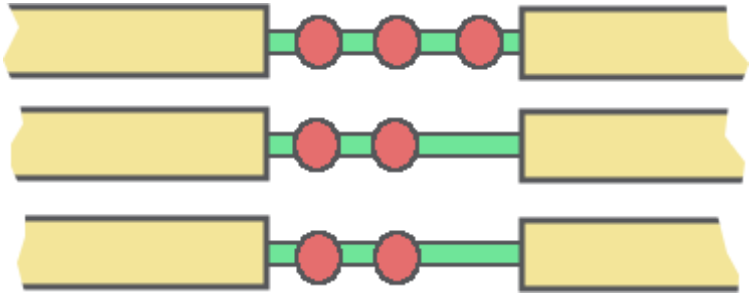
```
import math
def dot((x1, y1, z1), (x2, y2, z2)):
        """ Dot product of two 3-dimensional vectors"""
        return x1*x2 + y1*y2 + z1*z2

def normalize((x,y,z)):
        """ Normalizes a 3-dimensional vector"""
        len = math.sqrt(dot((x, y, z), (x, y, z)))
        return (x/len, y/len, z/len)

def angle(a, b):
        """ Finds an angle between two 3-dimensional vectors"""
        anorm = normalize(a)
        bnorm = normalize(b)
        return math.acos(dot(anorm, bnorm))
```

```
>>> from dot import *
>>> v1=(16, 16, 16)
>>> v2=(32, 32, 32)
>>> v3=(1,2,3)
>>> angle(v1, v3)
0.38759668665518016
>>> angle(v1, v2)
Traceback (most recent call last):
  File "", line 1, in
  File "dot.py", line 16, in angle
    return math.acos(dot(anorm, bnorm))
ValueError: math domain error
```

```
>>> dot(normalize(v1), normalize(v2))
1.0000000000000002
```

- Arccosine is only defined inside [-1, 1]
- Trying to take arccosine from 1.0000000000000002
- Result of accumulated rounding errors

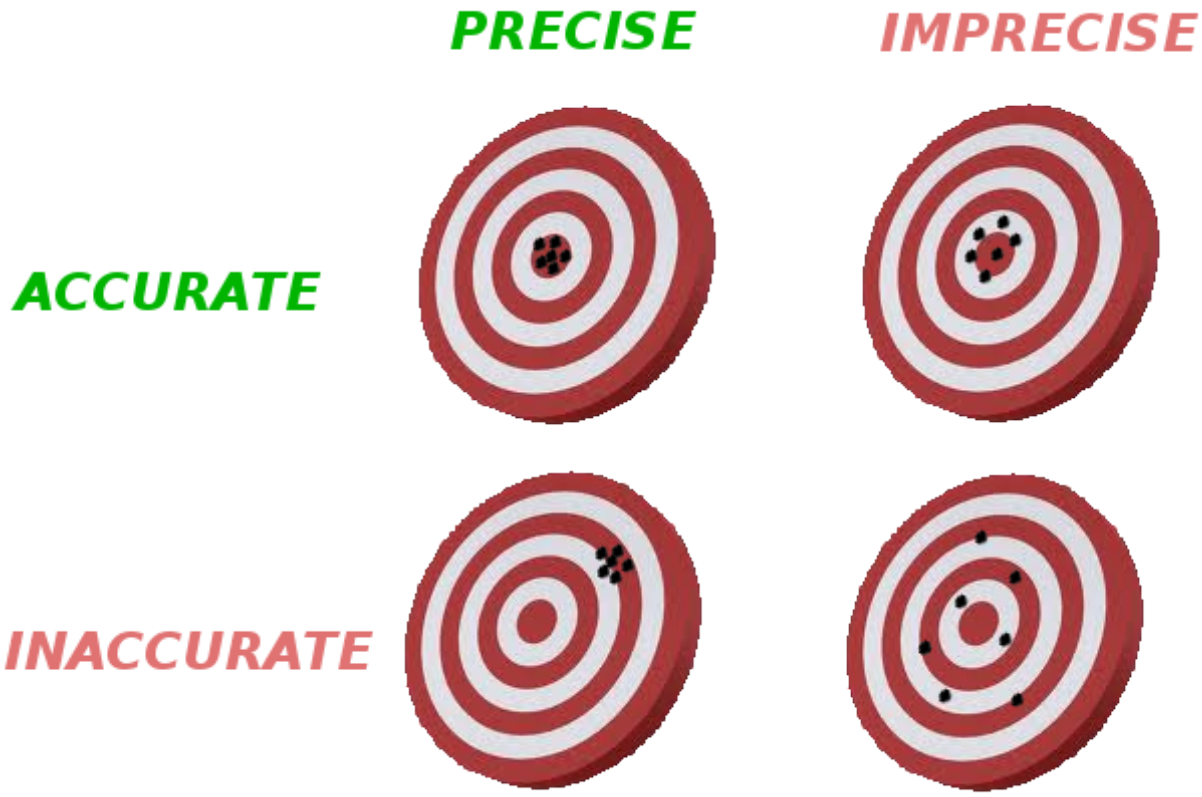
Problem: Bad conditioning

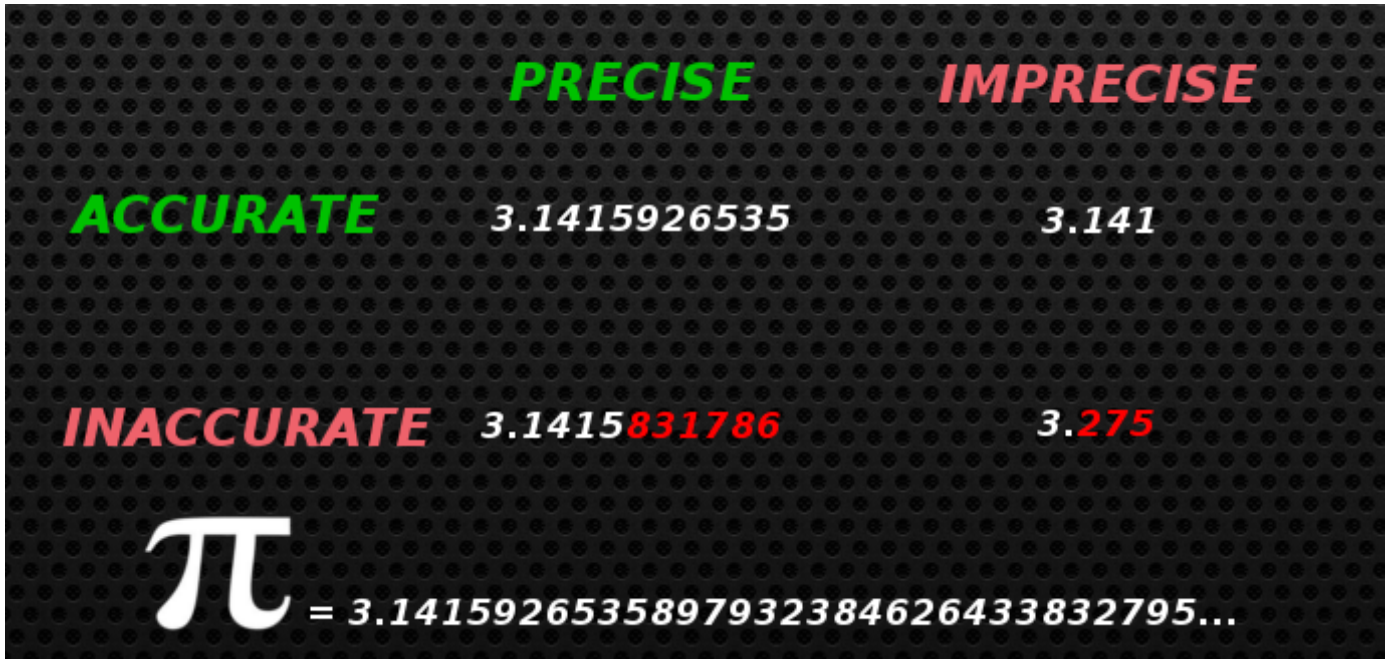


- Bad conditioning number: small changes in input cause big changes in output
- Can happen e.g. when going “continuous”->discrete via decimal rounding/truncation
- The problem: number of inserts (pink dots) defined from the (“constant”) length via rounding
- In one case: 2.499999786 -> 2, in other 2.50000000 -> 3
- Need different algorithm/architecture

Precision vs Accuracy

- Are not the same!

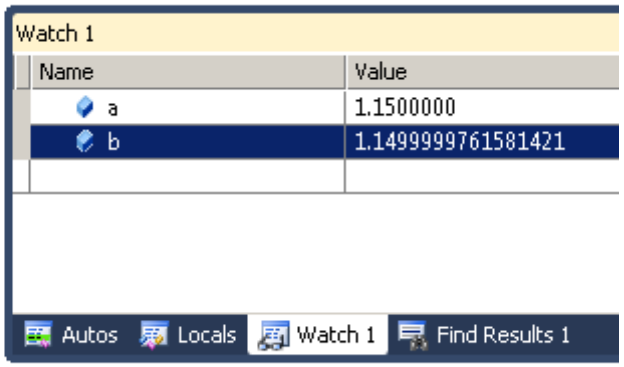




Double vs. Float, Precision vs. Accuracy

```c
int main() {
    float a;
    double b;

    a = 1.15f;
    b = a;

    printf("1.15 in float: %g\n", a);
    printf("1.15 in double: %g", b);
    return 0;
}
```

```
int main() {
    float a;
    double b;

    a = 1.15f;
    b = a;

    printf("1.15 in float: %g\n", a);
    printf("1.15 in double: %g", b);
    return 0;
}
```



> Double has worse accuracy than float???..

- 0.15 in single precision binary format:
  0 0 1 1 1 1 1 0 0 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 1 0 1 0 0 1 1 0 0 1

- 0.15 in double precision binary format:
  0 0 1 1 1 1 1 0 0 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1

- 0.15 in float->double precision binary:
  0 0 1 1 1 1 1 0 0 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

- The digits were chopped off!

- The printing code does “the right thing” for float, but truncated double has lost information

Floating point comparison

- Naive way that does not work:

> $a = b$

- You don’t just compare like that (0.1 + 0.2 != 0.3)

- Some programming languages (like ML) don’t even allow that

- A “better” way (does not really work either:

> $|a-b| < \epsilon$

- Which epsilon to choose?
- A better way: relative error

> $|a-b| < \epsilon \max(|a|,|b|)$

- Should also take NaNs into account, when applicable

- Still can be improved

- Can also compare as integers!

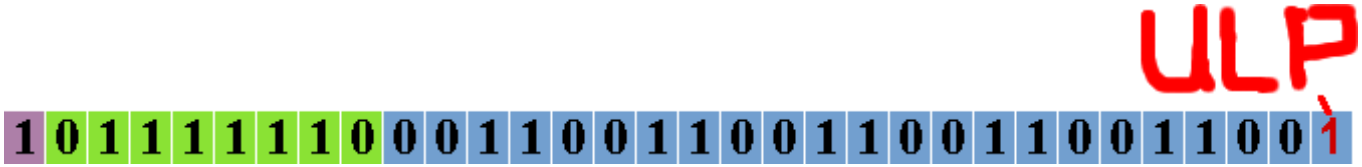
- Thanks to the way IEEE 754 representation is designed

- Machine epsilon: the distance between 1.0 and the next representable float number

- “Upper bound on the relative error”

> $meps_{32} = 2^{-24} = 5.96 \times 10^{-08}$
> $meps_{64} = 2^{-52} = 1.11 \times 10^{-16}$

- ULP (unit in the last place). Depends on the actual value.



> $ULP(x) = meps * 2^{E}$

Remedies: Use higher precision

- Sometimes a valid thing to do
- Would help with carrots planting
- However: might be just postponing/hiding a problem
- However: twice the memory usage
- However: worse SIMD utilization
- Compromise: do selectively, in the critical places

Anecdote! MMORPG server simulation time

- Anarchy Online by Funcom, released in 2001
- Massive Multiplayer Online Role Playing Game (MMORPG)
- Client-server architecture, all simulation run on server
- Initially used single-precision floating point for server time
- …crashed the server after a couple of weeks uptime

Remedies: Fixed point computation

- Work with integers and scale when required
- Reality for embedded microprocessors (no FPU anyway)
- Most common binary and decimal scales (10^X or 2^X)

```
#define NUM_CARROTS 10000        /* the number of carrotts to plant */
const long BOBWALK_WIDTH  = 480; /* the width of the bobwalk (0.01 feet) */
const long PLANT_INTERVAL = 125; /* the planting interval (0.01 feet) */

long carrot_pos[NUM_CARROTS];     /* carrot positions go here */

int main() {
    int i;
    /* Plant the carrots! */
    for (i = 0; i < NUM_CARROTS; i++) {
        carrot_pos[i] = BOBWALK_WIDTH + i*PLANT_INTERVAL;
        printf("Carrot %d goes to %d.%d feet!\n",
                       i, carrot_pos[i]/100, carrot_pos[i]%100);
    }
    return 0;
}
```

Remedies: Fixed point computation

```
void test_carrots() {
    /* verify that carrots are planted evenly! */
    int i, num_bad = 0;
    for (i = 1; i < NUM_CARROTS; i++) {
        long d = carrot_pos[i] - carrot_pos[i - 1];
        if (d != PLANT_INTERVAL) {
            printf("The carrot %d is planted wrong!!!\n", i);
            num_bad++;
        }
    }
    if (num_bad == 0) printf("All carrots placed well!!!");
    else printf( "%d of %d carrots placed incorrectly... :(" ,
```

```
                num_bad, NUM_CARROTS);
}
```

```
$ ./carrots
Carrot 0 goes to 4.80 feet!
Carrot 1 goes to 6.5 feet!
...
Carrot 3272 goes to 4094.80 feet!
Carrot 3273 goes to 4096.5 feet!
...
Carrot 9998 goes to 12502.30 feet!
Carrot 9999 goes to 12503.55 feet!

All carrots placed well!!!
```

Remedies: Rational Number Types

- Represent numbers as fractions, e.g. “0.1” is “1/10”
- Some programming languages (Clojure, Haskell, Mathematica, Perl), support natively
- Otherwise, libraries, e.g. GMP
- Clojure example:

```
user=> (def a 0.1)
user=> (def b 0.2)
user=> (def c 0.3)
user=> (= c (+ a b))
false

user=> (def ar (/ 1 10))
```

```
user=> (def br (/ 2 10))
user=> (def cr (/ 3 10))
user=> (= cr (+ ar br))
true

user=> [a b c]
[0.1 0.2 0.3]
user=> [ar br cr]
```

Remedies: Arbitrary-precision arithmetic

- Using as many digits of precision, “as needed”
- Theoretically limited by the system memory
- Usually quite sophisticated
- Trading correctness for performance
- Some programming languages support natively
- java.math.BigDecimal
- Libraries: Boost Multiprecision Library, MPFR, MAPM, TTMath
- Also, some standalone software (Maple, Mathematica)

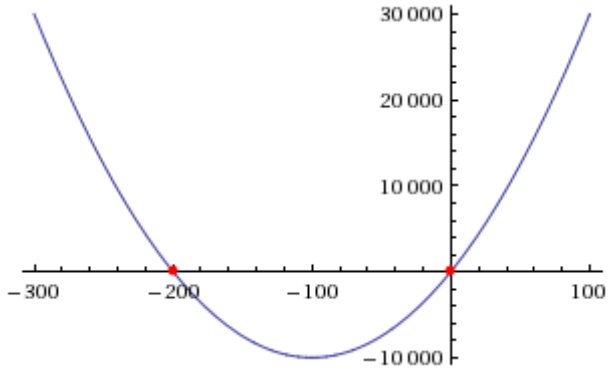
Remedies: Adapt the algorithms

- Quadratic Equation:

> $ax^2 + bx + c = 0$

- Example:

> $x^2 + 200x - 10.0025 = 0$



- Real roots: -200.05, 0.05

> $x^2 + 200x - 10.0025 = 0$

- Standard formula: can be inaccurate

> $$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

- Computed roots: -200.050003051758 , 0.0500030517578125, error: 3.05e-6
- An alternate formula to fight the catastrophfic cancellation:

> $$x_1 = \frac{-b - sign(b)\sqrt{b^2 - 4ac}}{2a}$$
> $$x_2 = \frac{c}{ax_1}$$

- Computed roots: -200.050003051758 , 0.0499999970197678, error: 3.7e-9

Improving accuracy: Number summation

```
/* Naive summation */
float naive_sum(float* val_arr, int nval) {
```

```
    int i;
    float acc = 0.0f;
    for (i = 0; i < nval; i++) acc += val_arr[i];
    return acc;
}
#define NUM_VAL 10000
int main() {
    int i;
    float val[NUM_VAL];
    for (i = 0; i < NUM_VAL; i++) val[i] = ((float)i + 1);

    printf("Naive sum: %f\n", naive_sum(val, NUM_VAL));
    return 0;
}
```

```
Naive sum: 50002896.000000
```

Improving accuracy: Kahan Summation

- Naive summation is a way off:
  $s = \frac{N*(1+N)}{2} = 50005000 \neq 50002896$

```
/* Kahan summation */
float kahan_sum(float* val_arr, int nval) {
    int i;
    float acc = 0.0f, y, t;
    float corr = 0.0f; /* running corrective value for rounding error*/
    for (i = 0; i < nval; i++) {
        y = val_arr[i] - corr; /* add the correction to the item */
        t = acc + y;           /* increase the sum, bits may be lost */
        corr = (t - acc) - y;  /* recover the lost bits*/
```

```
            acc = t;
        }
        return acc;
    }
```

```
Kahan sum: 50005000.000000
```

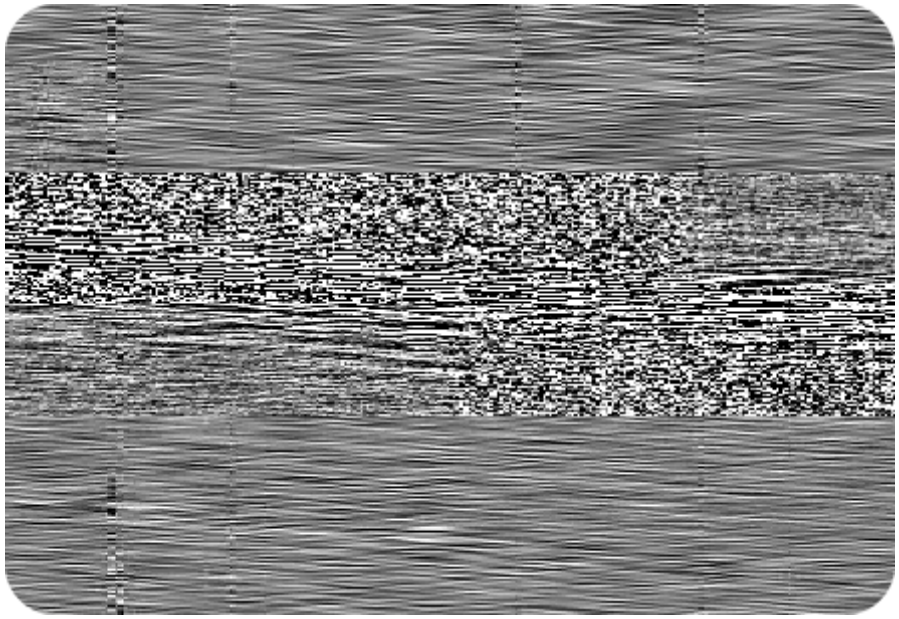
Floating Point Determinism

- Generally no guarantee that the same code would give the same results every time
- Things that might affect: hardware, OS, compiler, compilation flags, builds, 3rd party library behaviour
- Worse with multithreading
- Worse with x87
- Rounding/exception modes is a global per-thread setting
- But gets better with x64!

Anecdote! Online game synchronization

- Cossacks: European Wars by GSC Game World (2000)
- Real-Time Strategy Game (with online multiplayer)
- Run on Win32 PCs, different hardware
- Problems synchronizing
- …rewrote all the simulation in fixed point!

Anecdote! Intel vs GCC



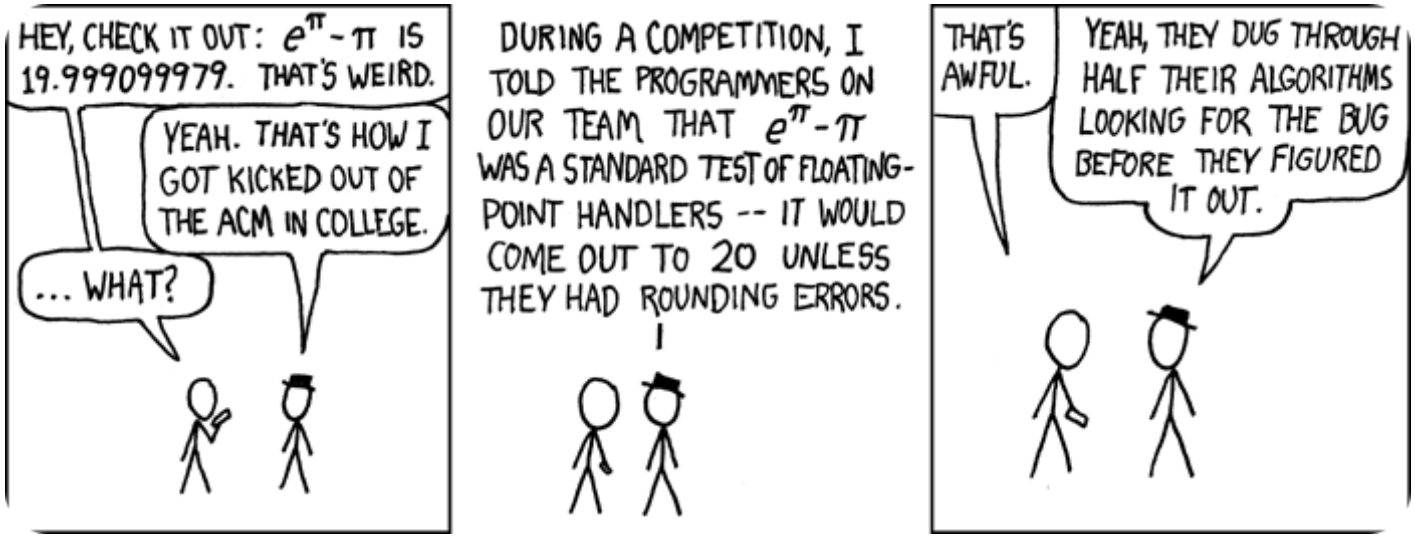
- Algorithm output matrix comparison
- The same C++ code, compiled with different compilers (GCC 4.7.1 and Intel Compiler)
- The scale is 10E-6. It's very close, but still not exactly the same.
- …can be other factors as well.

IEEE 754-2008, going forward

- IEEE 754 was introduced in 1985, quite long time ago
- Hardware capabilities, problems changed quite a bit
- IEEE 754-2008 is an upgrade of IEE 754 (backwards compatible!)
- Resolved minor ambiguities
- Extended with "half precision" (16-bit float) and "quad precision" (128-bit float)
- Arbirary precision formats

- Exception handling recommendations to the programming language implementers
- Fused Multiply-Add operation (A*B + C)
- Added transendental functions

Anecdote! Not a Floating Point fault



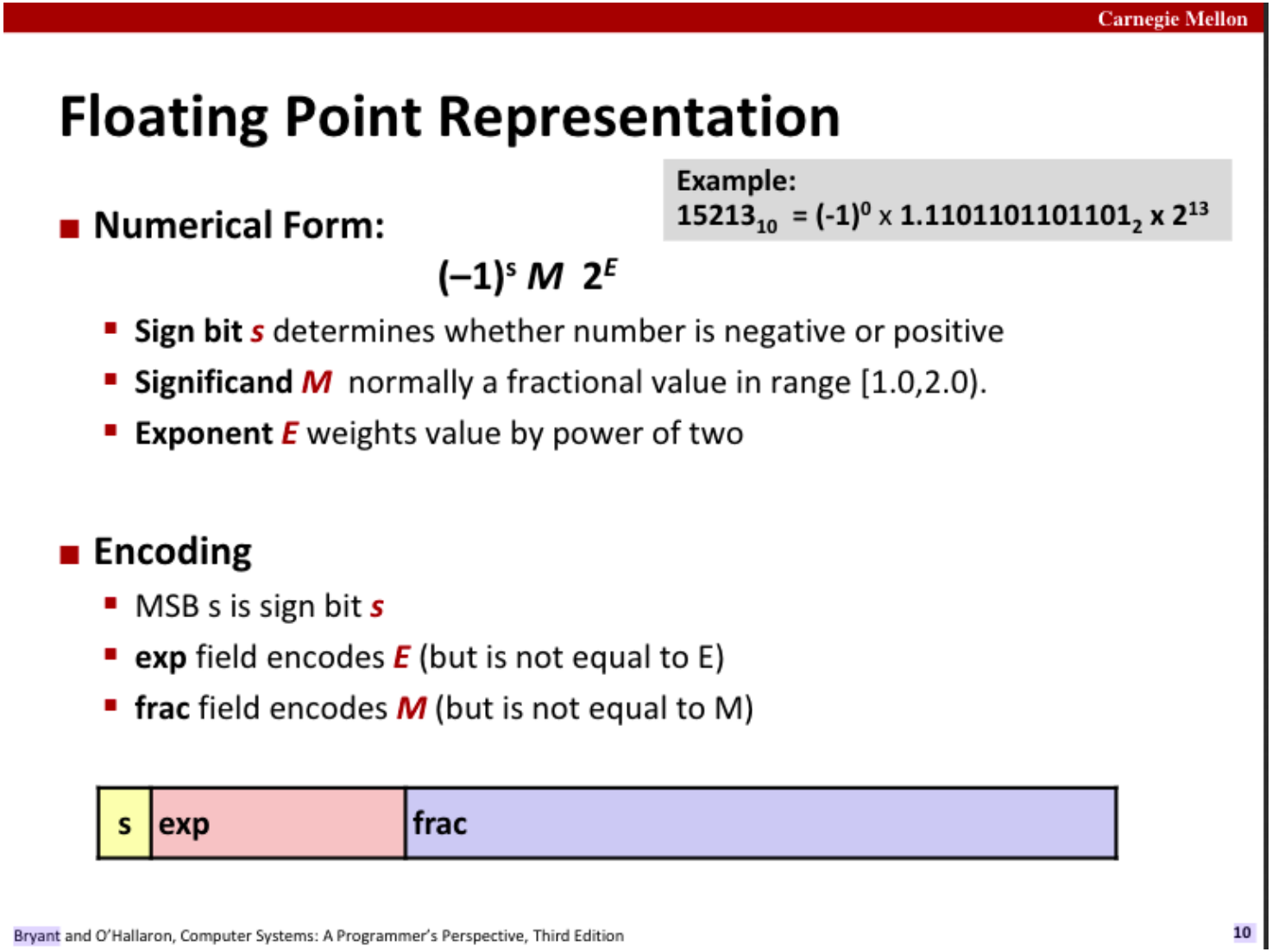
- or, “how to mess with people who’ve learned to expect rounding errors in floating-point math”

$e^{\pi} - \pi \approx 20$
$\pi^2 \approx \frac{227}{23}, \pi^3 \approx 31$

- …not everything explained with “rounding errors”!

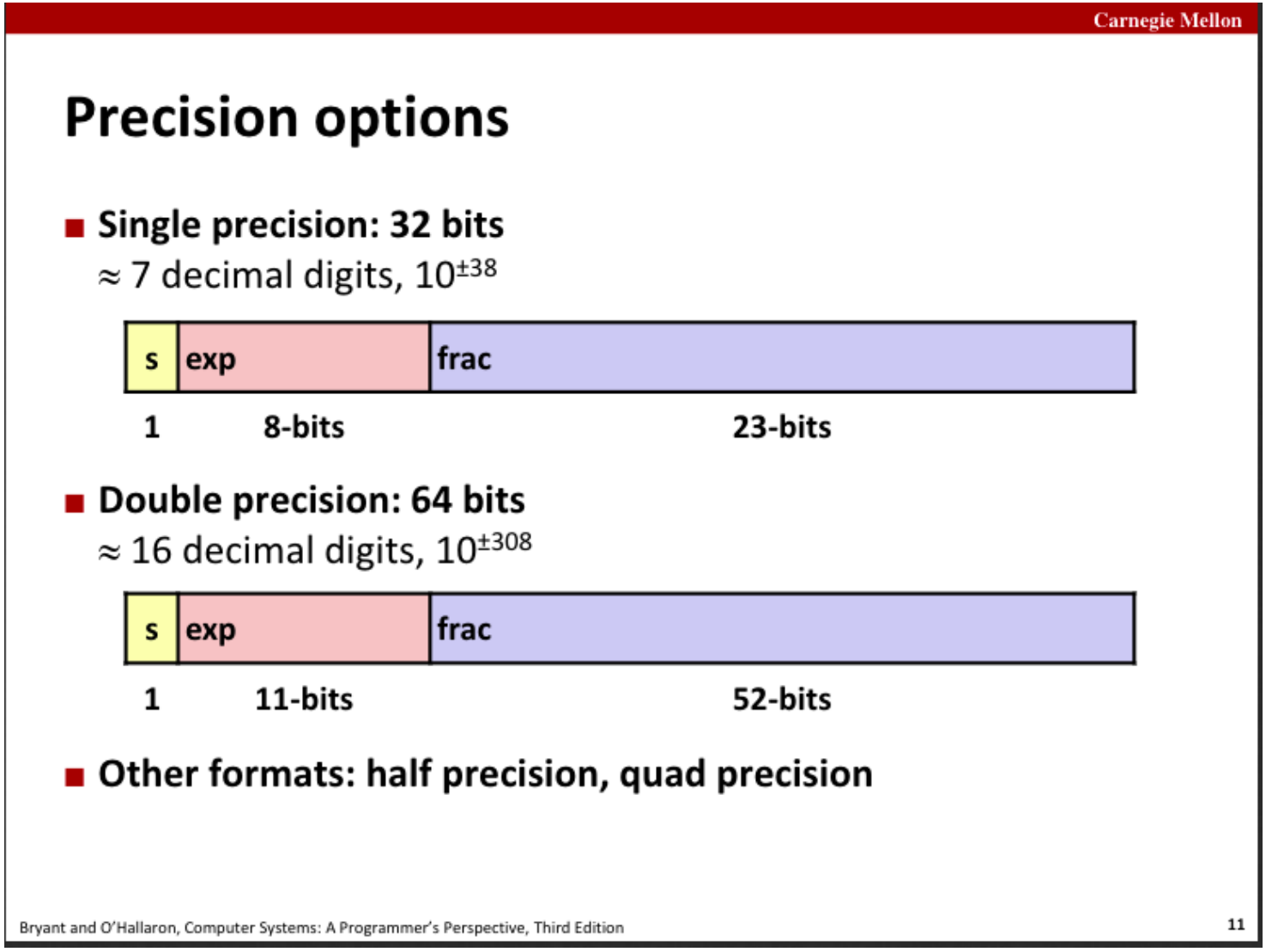
浮點數和定點數

- CMU ICS 錄影 / slides
  對應 CS:APP3e 的第 2 章: 2.4.2 IEEE 浮點數表示; 2.4.5 浮點運算



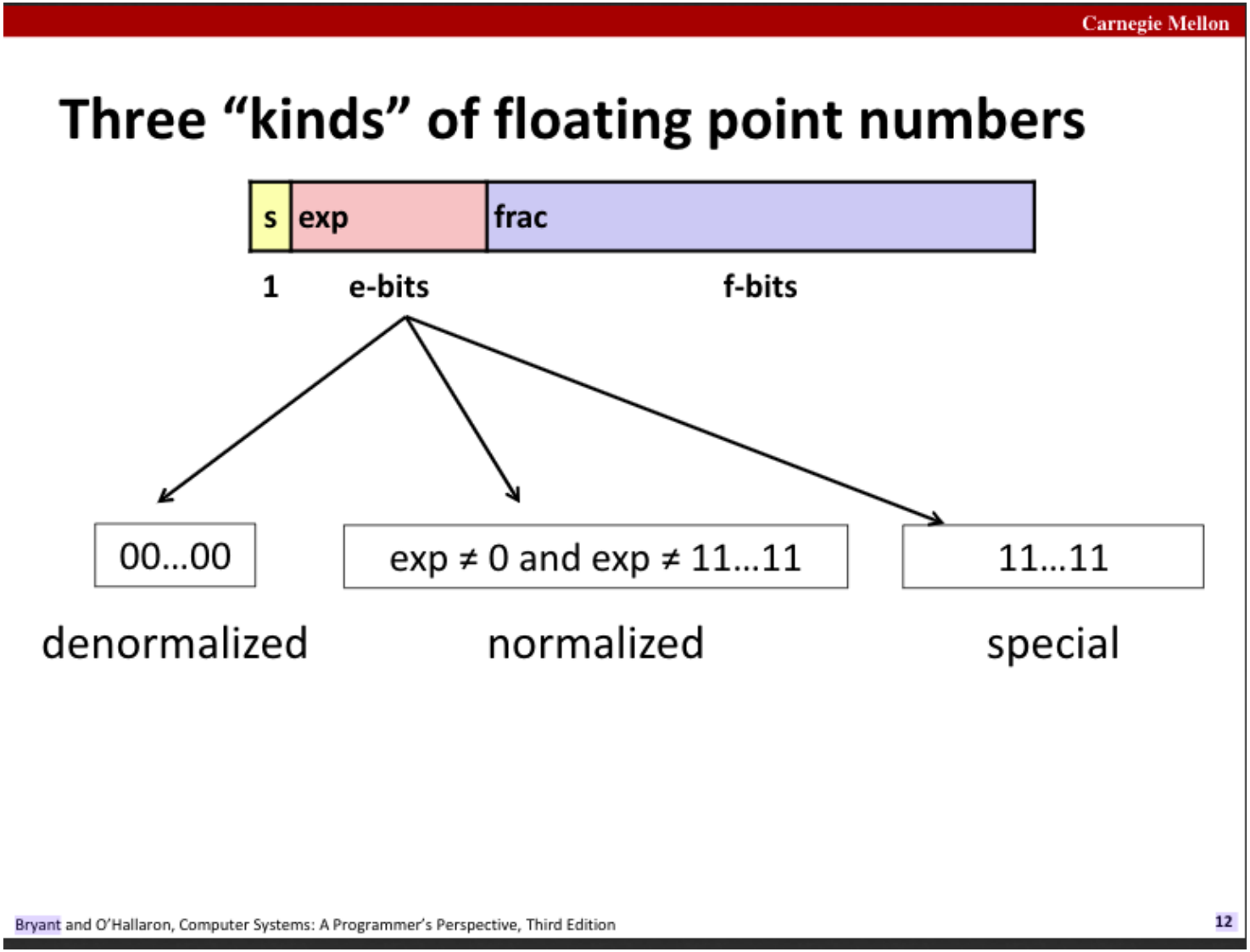
rounding error

- 電腦科學家制定如IEEE754之計算機標準，來表示實數家族中的分數，因為分數的數量是不可數的，電腦中浮點數儲存的是個近似值，使用上會產生誤差。



3 種浮點型態

- denormalized
- normalized
- special



“Normalized” Values

When: exp≠ 000…0 and exp≠ 111…1

Normalized Encoding Example

$$v = (-1)^s M 2^E$$
$$E = \mathbf{exp} - Bias$$

- **Value:** `float F = 15213.0;`
  - $15213_{10} = 11101101101101_2$
    $= 1.1101101101101_2 \times 2^{13}$
- **Significand**

  $M = 1.1101101101101_2$

  $\mathbf{frac} = 1101101101101\underline{0000000000}_2$
- **Exponent**

  $E = 13$

  $Bias = 127$

  $\mathbf{exp} = 140 = 10001100_2$
- **Result:**

0	10001100	11011011011010000000000
s	exp	frac

Denormalized Values

Condition: exp = 000…0

Exponent value: E= 1 –Bias (instead of exp–Bias)

Special Values

Condition: exp= 111…1

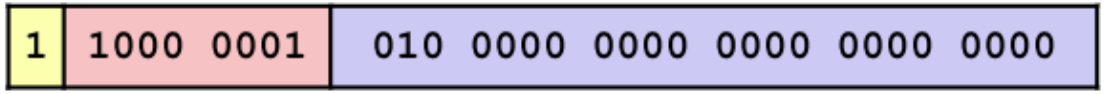
C float Decoding Example

$$v = (-1)^s M 2^E$$
$$E = \text{exp} - Bias$$

$Bias = 2^{k-1} - 1 = 127$

float: 0xC0A00000

binary: 1100 0000 1010 0000 0000 0000 0000 0000



1 8-bits 23-bits

E = exp – Bias = 129 – 127 = 2 (decimal)

S = 1 -> negative number

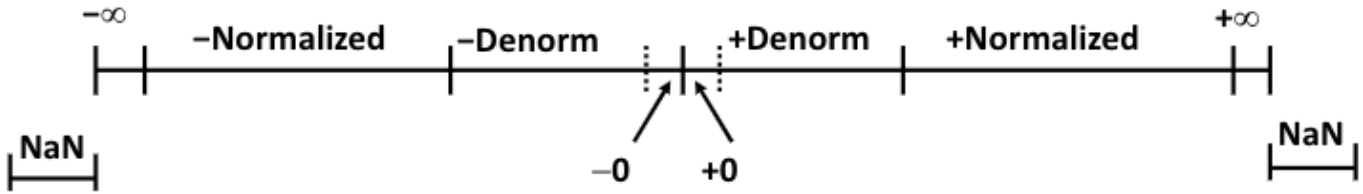
M = 1.010 0000 0000 0000 0000 0000

= 1 + 1/4 = 1.25

v = $(-1)^s$ M 2^E = $(-1)^1$ * 1.25 * 2^2 = -5

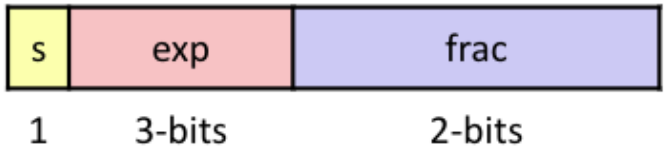
Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Visualization: Floating Point Encodings

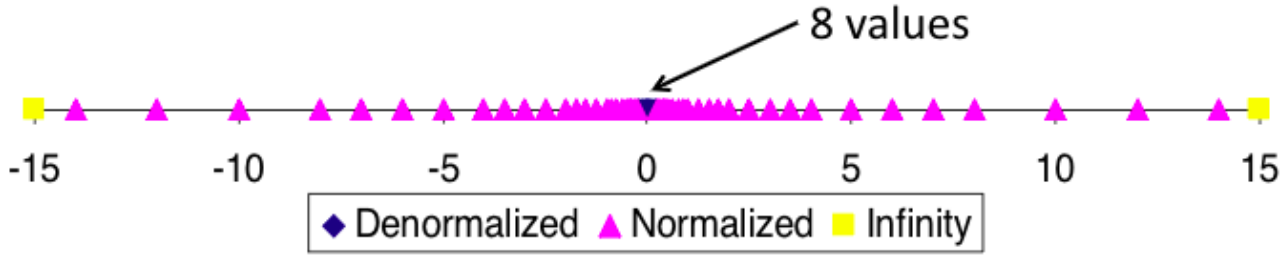


Distribution of Values

- **6-bit IEEE-like format**
  - e = 3 exponent bits
  - f = 2 fraction bits
  - Bias is $2^{3-1}-1 = 3$

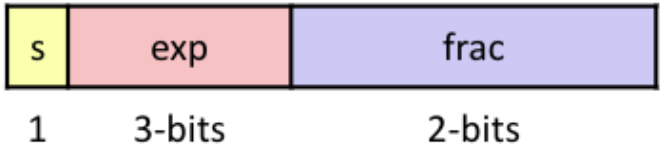


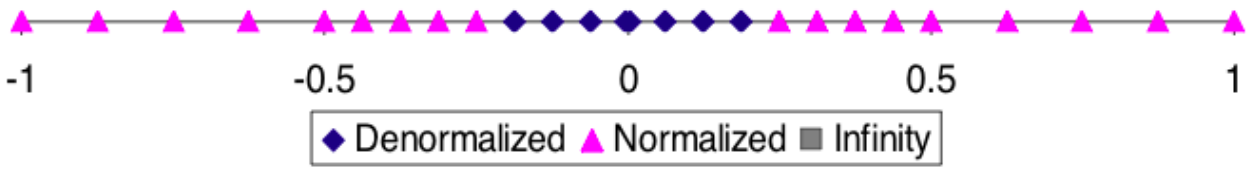
- **Notice how the distribution gets denser toward zero.**



Distribution of Values (close-up view)

- **6-bit IEEE-like format**
  - e = 3 exponent bits
  - f = 2 fraction bits
  - Bias is 3

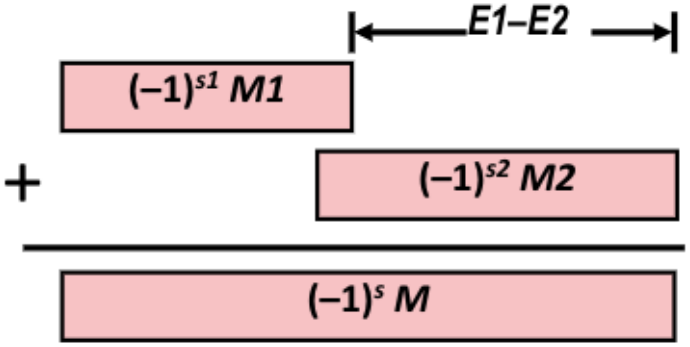




Floating Point Addition

- $(-1)^{s1}\ M1\ \ 2^{E1}\ \ +\ \ (-1)^{s2}\ M2\ \ 2^{E2}$
  - Assume $E1 > E2$
- **Exact Result: $(-1)^{s}\ M\ \ 2^{E}$**
  - Sign s, significand M:
    - Result of signed align & add
  - Exponent E: $E1$
- **Fixing**
  - If $M \geq 2$, shift M right, increment E
  - if $M < 1$, shift M left k positions, decrement E by k
  - Overflow if E out of range
  - Round M to fit `frac` precision

Get binary points lined up



$$1.010*2^2 + 1.110*2^3 = (0.1010 + 1.1100)*2^3$$
$$= 10.0110 * 2^3 = 1.00110 * 2^4 = 1.010 * 2^4$$

Mathematical Properties of FP Add

- **Compare to those of Abelian Group**
  - Closed under addition? *Yes*
    - But may generate infinity or NaN
  - Commutative? *Yes*
  - Associative? *No*
    - Overflow and inexactness of rounding
    - `(3.14+1e10)-1e10 = 0, 3.14+(1e10-1e10) = 3.14`
  - 0 is additive identity? *Yes*
  - Every element has additive inverse? *Almost*
    - Yes, except for infinities & NaNs
- **Monotonicity**
  - $a \geq b \Rightarrow a+c \geq b+c$? *Almost*
    - Except for infinities & NaNs

IEEE 754 浮點數值的比較，需要考慮到 sign + magnitude，實際操作類似以下:

```
#define FasI(f)   (*((int *) &(f)))
#define FasUI(f)  (*((unsigned int *) &(f)))
```

```
#define lt0(f)  (FasUI(f) > 0x80000000U)
#define le0(f)  (FasI(f) <= 0)
#define gt0(f)  (FasI(f) > 0)
#define ge0(f)  (FasUI(f) <= 0x80000000U)
```

- 浮點數運算和定點數操作

BFloat16

- 摘自 加速 AI 深度學習　BF16 浮點格式應運而生:

> 使用 BF16 格式時，由於指數小於 FP32，因而動態範圍大幅縮減。此外，將 FP32 數字轉換為 FP16 比起轉換為 BF16 更困難 —— 相較於僅截去尾數，FP16 更麻煩，而 BF16 的操作相對上較簡單。
> 另一個要點是計算所需要的晶片實體面積。由於硬體乘法器的實體尺寸會隨著尾數寬度的平方而增加，因此從 FP32 轉換到 BF16 可以大幅節省晶片面積 —— 這也就是 Google 之所以為其 TPU 晶片選擇使用 BF16。BF16 乘法器比 FP32 乘法器的尺寸更小 8 倍，而且也只有 FP16 同類型晶片約一半的尺寸。

待整理

- bfp - Beyond Floating Point - 探討 Posit Arithmetic
- Constant-Time Floating Point
- Integer Overflow/Underflow and Floating Point Imprecision
- Finding Root Causes of Floating Point Error
- What Every Computer Scientist Should Know About Floating-Point Arithmetic
- Basic Issues in Floating Point Arithmetic and Error Analysis

- 使用浮點數最最基本的觀念
- vf128 variable length floating-point
  > Android 的 ART/Dalvik 虛擬機器內部採用的 LEB128，也是變動長度的整數表示法
- 深入了解浮點數 IEEE 754
- Dive into floating point expressions
- Fixed-Point Arithmetic
