QCon

Please evaluate my talk via the mobile app!

INTERNATIONAL
SOFTWARE DEVELOPMENT
CONFERENCE

QCon

Catastrophic cancellation: the pitfalls of floating point arithmetic

(and how to avoid them!)

Graham Markall

Quantitative Developer, OpenGamma

Intro/Disclaimers

- Aims:
  - Get a rough "feel" of how floating point works
  - Know when to dig deeper
  - Cover basics, testing, and optimisation

- Not an exhaustive trudge through algorithms + details
- This talk: IEEE754 – mostly, but not quite ubiquitous
- Mostly C/Python examples, Linux/x86_64
- No complex numbers
- Code samples available!

Code ex. 1

A problem (C)

```
#include <values.h>

float a = MAXINT;   //           2147483648
float b = MAXLONG;  // 9223372036854775808
float f = a + b;

f == MAXLONG; // True or false?
```

Code ex. 1

A problem (C)

```
#include <values.h>

float a = MAXINT;    //           2147483648
float b = MAXLONG;   // 9223372036854775808
float f = a + b;

f == MAXLONG; // True or false?
// True!
// IEEE754 only approximates real arithmetic
```

How is arithmetic on reals approximated?

```
// float gives about 7 digits of accuracy
                             *******---.------
 MAXINT:                     2147483648.000000
MAXLONG:            9223372036854775808.000000
                    *******-------------.------
//                    ^            ^
//                    |            |
//                Represented   “Lost” beneath
//                                 unit of
//                              least precision
```

Floating point representation (1)

Sign – exponent – mantissa

$$s \text{ mantissa} * 2^{\text{exponent}}$$

Sign bit: 0 = positive, 1 = negative

Mantissa: 1.xxxxxxxxxx…

FP representation (2)

```
Mantissa has:
    implied leading 1
Exponent has:
    bias (-127 for float)
```

S|EEEEEEEE|1|MMMMMMMMMMMMMMMMMMMMMMM

```
// MAXINT:  S = 0, M = 1.0, E = 158
```

0|10011110|1|00000000000000000000000

$+ 1.0 * 2^{158-127} = 2147483648.0$

FP representation (2)

```
Mantissa has:
    implied leading 1
Exponent has:
    bias (-127 for float)
```

```
S|EEEEEEEE|1|MMMMMMMMMMMMMMMMMMMMMMM

// MAXINT:  S = 0, M = 1.0, E = 158

0|10011110|1|00000000000000000000000

+ 1.0 * 2^(158-127) = 2147483648.0

// MAXLONG: S = 0, M = 1.0, E = 190

0|10111110|1|00000000000000000000000

+ 1.0 * 2^(190-127) = 9223372036854775808.0
```

Code ex. 2

MAXLONG smallest increment

```
// MAXLONG: S = 0, M = 1.0, E = 190

0|10111110|1|00000000000000000000000

+ 1.0 * 2^(190-127) = 9223372036854775808.0

0|10111110|1|00000000000000000000001

+ 1.0000001192092896 * 2^(190-127) =
                              9223373136366403584.0
```

On the number line

MAXLONG + MAXINT
(0.19% towards MAXLONG_NEXT)

MAXLONG (9223372036854775808.0)

MAXLONG_NEXT (9223373136366403584.0)

Code ex. 3

Precision and range summary

- **Precision**: Mantissa length
- **Range**: Exponent length

- **Float**, 4 bytes:
  23 bit mantissa, 8 bit exponent
  Precision: ~7.2 digits
  Range: 1.17549e-38, 3.40282e+38

- **Double**, 8 bytes:
  52 bit mantissa, 11 bit exponent
  Precision: ~15.9 digits
  Range: 2.22507e-308, 1.79769e+308

Special cases

When is a number not a number?

Code ex. 4, 5

Floating point closed arithmetic

- Integer arithmetic:
  - `1/0   // Arithmetic exception`

- Floating point arithmetic is closed:
- Domain (double):
  - 2.22507e-308 <-> 1.79769e+308
  - 4.94066e-324 <-> just beneath 2.22507e-308
  - +0, -0
  - Inf
  - NaN
- Exceptions are exceptional – traps are exceptions

Code ex. 6

A few exceptional values

```
1/0 = Inf                // Limit
-1/0 = -Inf              // Limit
0/0 = NaN                // 0/x = 0, x/0 = Inf
Inf/Inf = NaN            // Magnitudes unknown
Inf + (-Inf) = NaN       // Magnitudes unknown
0 * Inf = NaN            // 0*x = 0, Inf*x = Inf
sqrt(x), x<0 = NaN       // No complex
```

Code ex. 7

Consequences

```
// Inf, NaN propagation:

double n = 1000.0;
for(double i = 0.0; i < 100.0; i += 1.0)
  n = n / i;
printf(“%f”, n); // “Inf”
```

Code ex. 8-12

Trapping exceptions (Linux, GNU)

- `feenableexcept(int __excepts)`
  - `FE_INXACT` – Inexact result
  - `FE_DIVBYZERO` - Division by zero
  - `FE_UNDERFLOW` - Underflow
  - `FE_OVERFLOW` - Overflow
  - `FE_INVALID` - Invalid operand
- SIGFPE Not exclusive to floating point:
  - `int i = 0; int j = 1; j/i // Receives SIGFPE!`

Back in the normal range

Some exceptional inputs
to some math library functions
result in normal-range results:

```
x = tanh(Inf)      // x is 1.0
y = atan(Inf)      // y is pi/2
```

(ISO C / IEEE Std 1003.1-2001)

Denormals

- x – y == 0 implies x == y ?
- Without denormals, this is not true:
  - X = 2.2250738585072014e-308
  - Y = 2.2250738585072019e-308 // (5e-324)
  - Y – X = 0
- With denormals:
  - 4.9406564584124654e-324
- Denormal implementation e = 0:
  - Implied leading 1 is not a 1 anymore
- Performance: revisited later

Testing

Getting *getting right* right

Assumptions

- Code that does floating-point computation
- Needs tests to ensure:
  - Correct results
  - Handling of exceptional cases
- A function to compare floating point numbers is required

Exact equality (danger)

```
def equal_exact(a, b):
    return a == b

equal_exact(1.0+2.0, 3.0)          # True

equal_exact(2.0, sqrt(2.0)**2.0) # False

sqrt(2.0)**2  # 2.0000000000000004
```

Code ex. 16

Absolute tolerance

```
def equal_abs(a, b, eps=1.0e-7):
    return fabs(a - b) < eps

equal_abs(1.0+2.0, 3.0)                 # True

equal_abs(2.0, sqrt(2.0)**2.0)          # True
```

Code ex. 17

Absolute tolerance **eps** choice

```
equal_abs(2.0, sqrt(2)**2, 1.0e-16) # False

equal_abs(1.0e-8, 2.0e-8)           # True!
```

Code ex. 18

Relative tolerance

```
def equal_rel(a, b, eps=1.0e-7):
    m = min(fabs(a), fabs(b))
    return (fabs(a – b) / m) < eps

equal_rel(1.0+2.0, 3.0)                  # True
equal_rel(2.0, sqrt(2.0)**2.0)           # True
equal_rel(1.0e-8, 2.0e-8)                # False
```

Code ex. 19

Relative tolerance correct digits

eps	**Correct digits**
1.0e-1	~1
1.0e-2	~2
1.0e-3	~3
…	
1.0e-16	~16

Relative tolerance near zero

```
equal_rel(1.0e-50, 0)

ZeroDivisionError: float division by zero
```

Summary guidelines:

When to use:

- Exact equality: **Never**
- Absolute tolerance: **Expected ~ 0.0**
- Relative tolerance: **Elsewhere**

- Tolerance choice:
  - No universal “correct” tolerance
  - Implementation/application specific

- Appropriate range: application specific

Code ex. 21

Checking special cases

```
 -0  ==    0  // True

 Inf ==  Inf  // True
-Inf == -Inf  // True

 NaN ==  NaN  // False
 Inf ==  NaN  // False

 NaN  < 1.0   // False
 NaN  > 1.0   // False
 NaN == 1.0   // False
 isnan(NaN)   // True
```

Performance optimisation

Manual and automated.

Division vs Reciprocal multiply

```
// Slower (generally)
a = x/y;           // Divide instruction

// Faster (generally)
y1 = 1.0/y;        // x86: RCPSS instruction
a = x*y1;          // Multiply instruction

// May lose precision.
// GCC: -freciprocal-math
```

Non-associativity

```
float a = 1.0e23;
float b = -1.0e23;
float c = 1.0;
printf("(a + b) + c = %f\n", (a + b) + c);
printf("a + (b + c) = %f\n", a + (b + c));

(a + b) + c = 1.000000
a + (b + c) = 0.000000
```

Non-associativity (2)

- Re-ordering is "unsafe"
- Turned off in compilers by default
- Enable (gcc):
  `-fassociative-math`
- Turns on `–fno-trapping`, also `–fno-signed-zeros` (may affect `-0 == 0`, flip sign of `-0*x`)

Finite math only

- Assume that no Infs or NaNs are ever produced.
- Saves execution time: no code for checking/dealing with them need be generated.

- GCC: `-ffinite-math-only`

- Any code that uses an Inf or NaN value will probably behave incorrectly
  - This can affect your tests! `Inf == Inf` may not be true anymore.

-ffast-math

- Turns on all the optimisations we've just discussed.
- Also sets flush-to-zero/denormals-are-zero
  - Avoids overhead of dealing with denormals
  - x – y == 0 -> x == y may not hold

- For well-tested code:
  - Turn on –ffast-math
  - Do tests pass?
  - If not, break into individual flags and test again.

-ffast-math linkage

- Also causes non-standard code to be linked in and called
- e.g. `crtfastmath.c set_fast_math()`
- This can cause havoc when linking with other code.

- E.g. Java requires option to deal with this:
- `-XX:RestoreMXCSROnJNICalls`

Summary guidelines

- Refactoring and reordering of floating point can increase performance
- Can also be unsafe
- Some transformations can be enabled by compiler
- Manual implementation also possible

- Make sure code well-tested
- Be prepared for trouble!

Wrap up

Floating point

- Finite approximation to real arithmetic
- Some “corner” cases:
  - Denormals, +/- 0
  - Inf, NaN
- Testing requires appropriate choice of:
  - Comparison algorithm
  - Expected tolerance and range
- Optimisation:
  - For well-tested code
  - Reciprocal, associativity, disable “edge case” handling

- FP can be a useful approximation to real arithmetic

Please evaluate my talk via the mobile app!

Code samples/examples:

https://github.com/gmarkall/PitfallsFP

INTERNATIONAL SOFTWARE DEVELOPMENT CONFERENCE