Scientific Programming Techniques

Summer school of Parallel Computing at Cineca
Bologna, June 6-17 2016

Outline

1. Floats are not real
   a. Rounding is all around
   b. Optimize like a boss (for real)
   c. Compare with care
2. Ignoring exceptions
   a. Being brave with exceptions
   b. Compiler is your friend (fortran)
   c. Compiler is (NOT) your friend (C/C++)

Outline

1. Floats are not real
   a. Rounding is all around
   b. Optimize like a boss (for real)
   c. Compare with care
2. Ignoring exceptions
   a. Being brave with exceptions
   b. Compiler is your friend (fortran)
   c. Compiler is (NOT) your friend (C/C++)

One is not one

```
#include <iostream>
#include <iomanip>
int main(){
  float a = 0.1f;
  float sum_a, mul_a = 0.0f;
  for (int i=0; i<10; ++i){
    sum_a += a;
  }
  mul_a = a*10.0f;
  std::cout << std::scientific << std::setprecision(9);
  std::cout << "Summing ten times gives: " << sum_a << std::endl;
  std::cout << "Multiplying by ten gives: " << mul_a << std::endl;
  return 0;
}
```

One is not one

Compile:

```
$ g++ test_one.cpp -o test_one.x

$ icpc test_one.cpp -o test_one.x
```

One is not one

Compile:

```
$ g++ test_one.cpp -o test_one.x
$ icpc test_one.cpp -o test_one.x
```

Run (both):

```
$ ./test_one
Summing ten times gives: 1.000000119e+00
Multiplying by ten gives: 1.000000000e+00
```

One is not one (reloaded)

```
$> python3
Python 3.3.0 (default, Sep 19 2013, 14:43:35)
[GCC 4.7.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> .1 + .1 + .1 == .3
False
>>> round(.1,1) + round(.1,1) + round(.1,1) == round(.3,1)
False
>>> round(.1 + .1 + .1, 1) == round(.3, 1)
True
>>> from decimal import Decimal
>>> Decimal.from_float(0.1)
Decimal('0.1000000000000000055511151231257827021181583404541015625')
```

One is not one (reloaded)

```
$> python3
Python 3.3.0 (default, Sep 19 2013, 14:43:35)
[GCC 4.7.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> .25 + .25 == .5
True
>>> round(.25, 2) + round(.25, 2) == round(.5, 2)
True
>>> round(.25 + .25, 2) == round(.5, 2)
True
>>> from decimal import Decimal
>>> Decimal.from_float(.25)
Decimal('0.25')
```

2.675 ROUNDS STRANGE

```
$> python3
Python 3.3.0 (default, Sep 19 2013, 14:43:35)
[GCC 4.7.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> round(2.675, 2)
2.67
>>> round(1.675, 2)
1.68
>>> from decimal import Decimal
>>> Decimal.from_float(2.675)
Decimal('2.67499999999999982236431605997495353221893310546875')
>>> Decimal.from_float(1.675)
Decimal('1.67500000000000004440892098500626161694526672363281 25')
```

Floats are not real

```
#include <iostream>
#include <iomanip>
int main(){
float x = 1.000e30;
float y = -1.000e30;
float z = 1.0f;
float a,b,c;
a=(x+y)+z;
b=x+(y+z);
c=x+y+z;
std::cout << std::scientific << std::setprecision(8);
std::cout << "a= " << a << " b= " << b << " c = " << c << std::endl;
}
```

Floats are not real

Compile:

```
$ g++ -O2 test_associative.cpp -o test.x
$ icpc -O0 test_associative.cpp -o test.x
```

Floats are not real

Compile:

```
$ g++ -O2 test_associative.cpp -o test.x
$ icpc -O2 test_associative.cpp -o test.x
```

Run (both):

```
$ ./test.x
a= 1.00000000e+00 b = 0.00000000e+00 c = 1.00000000e+00
```

Floats are not real

```
#include <iostream>
#include <iomanip>
#include <cmath>

int main(){
std::cout << std::scientific << std::setprecision(9) << std::nextafter(1.e30f + 1.f, INFINITY) << std::endl ;
std::cout << (1.e30f + 1.f == 1.e30f) << std::endl;
return 0;
}
```

```
./example.x
1.000000091e+30
1
```

Epic Fail

```
#include <iostream>
#include <iomanip>
#include <cmath>
int main(){
const float x = 1.1e-08f;
const double dx = 1.1e-08;
const float fcos = cos(1.1e-08f);
const double dfcos = cos(1.1e-08);
float ratio = (1.f-fcos)/(x*x);
double dratio  = (1.-dfcos)/(dx*dx);

std::cout << std::scientific << std::setprecision(8) << "Float gives " << ratio << std::endl;
std::cout << std::scientific << std::setprecision(8) << "Double gives " << dratio << std::endl;

return 0;
}
```

Epic Fail

Compile:

```
$ g++ -O2 test_epic.cpp -o test.x
$ icpc -O2 test_epic.cpp -o test.x
```

Epic Fail

Compile:

```
$ g++ -O2 test_epic.cpp -o test.x
$ icpc -O2 test_epic.cpp -o test.x
```

Run (both):

```
$ ./test.x
Float gives 0.00000000e+00
Double gives 9.17539690e-01 (and exact is almost .5)
```

Outline

1. Floats are not real
    a. Rounding is all around
    b. Optimize like a boss (for real)
    c. Compare with care
2. Ignoring exceptions
    a. Being brave with exceptions
    b. Compiler is your friend (fortran)
    c. Compiler is (NOT) your friend (C/C++)

The great optimizer

```
#include <iostream>
#include <iomanip>
int main(){
   float c=1.e30f;
   float d=1.e-31f;
   float optim;
   std::cout << std::scientific << std::setprecision(5);
   float val = 1.e30f/c/d;
   optim=1.0f/(c/d);
   float val_opt = 1.e30f * optim;
   std::cout << "Value is " << val << std::endl;
   std::cout << "Optimized value is " << val_opt << std::endl;
}
```

The great optimizer

Compile:

```
$ g++ -O2 test_optimizer.cpp -o test.x
$ icpc -O2 test_optimizer.cpp -o test.x
```

The great optimizer

Compile:

```
$ g++ -O2 test_optimizer.cpp -o test.x
$ icpc -O2 test_optimizer.cpp -o test.x
```

Run (both):

```
$ ./test.x
Value is 1.00000e+31
Optimized value is 0.00000e+00
```

Outline

1. Floats are not real
   a. Rounding is all around
   b. Optimize like a boss (for real)
   c. Compare with care
2. Ignoring exceptions
   a. Being brave with exceptions
   b. Compiler is your friend (fortran)
   c. Compiler is (NOT) your friend (C/C++)

Compare with care

```
#include <iostream>
#include <limits>
#include <cmath>
int main() {
  using namespace std;
  float onetenth = 1./10.;
  if (onetenth == 0.1) {
    std::cout << "LIFE IS WONDERFUL" << std::endl;
  }
  else {
    std::cout << "Ops!" << std::endl;
  }
  std::cout << "Isequal gives: " << isequal(onetenth, 0.1) << std::endl;
  std::cout << "Same gives: " << same(onetenth, 0.1) << std::endl;
}
```

Compare with care

```
bool isequal(float a, float b){
   float epsilon =   std::numeric_limits<float>::
epsilon();
   float absA = std::abs(a);
   float absB = std::abs(b);
   float diff = std::abs(a-b);
   if (a == b){
       std::cout << "case: exactly equal" << std::
endl;
       std::cout.flush();
       return 1;
   }
   else if(a*b == 0){
        std::cout << "case: one is zero" << std::
endl;
        return diff < (epsilon*epsilon);
   }
   else {
       std::cout << "case: not zero and not equal" <<
std::endl;
       float largest=(absB > absA) ? absB : absA;
       return diff < largest*epsilon;
   }
}
```

Compare with care

```
bool same(float a, float b)
{
  return std::nextafter(a, std::numeric_limits<float>::lowest()) <= b
    && std::nextafter(a, std::numeric_limits<float>::max()) >= b;
}
```

Compare with care

Compile:

```
$ g++ -O2 test_compare.cpp -o test.x
$ icpc -O2 test_compare.cpp -o test.x
```

Compare with care

Compile:

```
$ g++ -O2 test_compare.cpp -o test.x
$ icpc -O2 test_compare.cpp -o test.x
```

Run (both):

```
$ ./test.x
Ops!
case: exactly equal
Isequal gives: 1
Same gives: 1
```

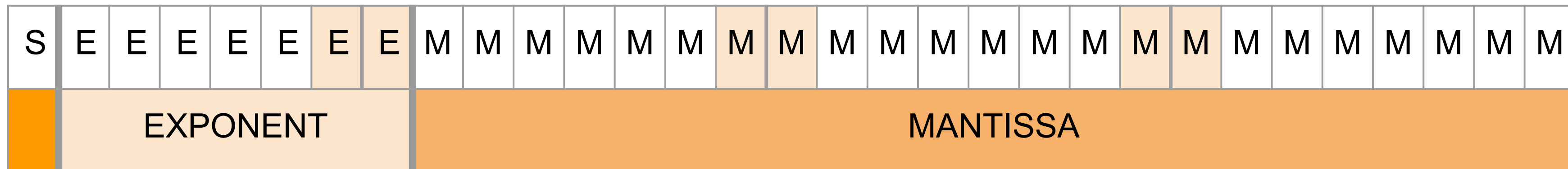
Representing Real Numbers

1/3 ≈ $0.333333[....]_{10}$ since it cannot be expressed as a sum of powers of 10

1/10 ≈ $0.00011001100110011001100110011[...]_2$ since it cannot be expressed as a sum of powers of 2

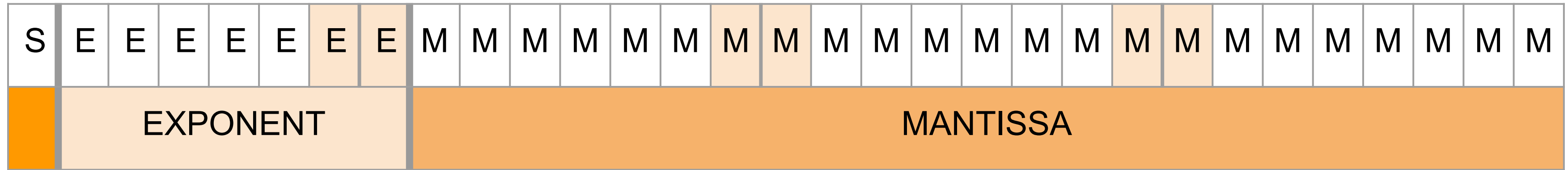
1/4 = 0.01_2 and its binary representation is exact

Representing Real Numbers



- IEEE 754: Floats
  - 32 bits: 1(sign), 8 exponent, 23 (+1) significand
  - $[(-1)S] * [2^{(E-127)}] * 1.M$, $0 < E < 255$
- IEEE 754: Double
  - 64 bits: 1(sign), 11 exponent, 52 (+1) significand
  - $[(-1)S] * [2^{(E-1023)}] * 1.M$, $0 < E < 2047$

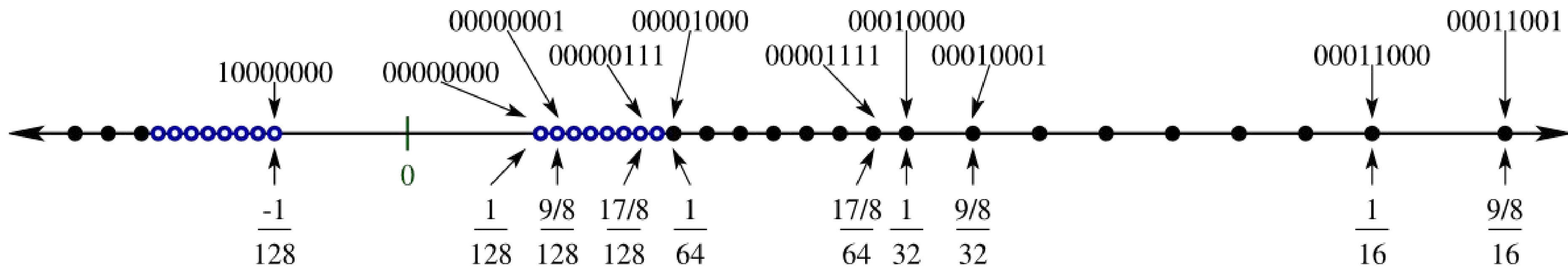
Representing Real Numbers



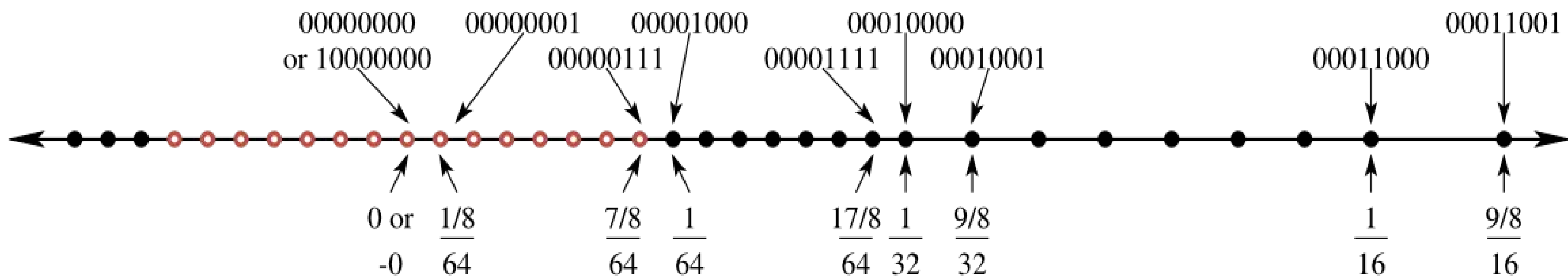
- IEEE 754: Floats
  - Norms: Range: $1.17549435E^{-38} < |f| < 3.4028235E^{+38}$
  - Norms Precision: 7 (decimal) digits
  - Denorms: Range: $1.4E^{-45} < |f| < 1.1754942E^{-38}$
  - Denorms: Precision 6 (decimal) digits.

- IEEE 754: Double
  - $2.2250738585072020x10E^{-308} < |f| < 1.797693134862315 7x10E^{+308}$
  - Precision 15 (decimal) digits.

Floats Float AROUND



Floats Float AROUND



1.23, 1.24, 1.235

Example: one (decimal) digit representation

All you never dared to ask about epsilon

Epsilon

* The relative error is bounded by machine ε $2^{-24}(5.96 \times 10^{-8})$
* The absolute error for a float number f = d.dd[...] * 2^E is bounded by ε*2^E

Floating floats

* Floats are not equally spaced, since the minimum spacing increases as the exp increases.
* Spacing between f and the next float is given by $2 \times \varepsilon \times 2^E$, 1 **ulp**

IEEE and precision

* IEEE asks the representation of floats to be exact within 0.5 ulps (exact rounding)
* IEEE asks +, -, ×, /, sqrt, remainder, internal to decimal to be exact rounded. Trascendental functions are computed within 0.5 to 1 ulp

LIVING WITHOUT ULPS

Cancellation is odd

* Sum and subtractions can suffer catastrophic cancellations.
* Least significant bits can become most significant bits, little differences can become 0.
* Beware of near numbers subtractions

LIVING WITHOUT ULPS

Compare with care

* Two numbers differing by a quantity < 1 ulp are equal, but ulp value changes with exponent.
* Take it into account when comparing two floats: == is not the safe choice.

LIVING WITHOUT ULPS

Rounding is all around

* Rounding error propagate. Apply propagation rules for maximum relative error
* … and try to normalize physical quantities
* … and beware of unstable algorithms: rounding off errors magnify problems

Compiler is your friend

Ask your compiler to be IEEE 754 precise.

- Gnu: g++/gcc/gfortran
  -O0 -g -frounding-math -fsignaling-nans (IEEE compliance)
- Intel:icc/icpc/ifort
  -O0 -g -fp-model strict

Ask your compiler to generate fast code

- Gnu: gcc/gfortran -O2/O3 -ffast-math -ftz
- Intel: icc/ifort -O2/O3 -fp-model fast, -no-prec-div , -no-prec-sqrt, -ftz

In Brief:

It’s fast, but it’s wrong

* Disable optimizations: -O0 - g(for debugging).
* Use precision flags.
* Devise a small test case and perform calculations in double precision.
* Only if the algorithm is numerically stable.

It’s slow, but it’s right

* Enable optimizations: -O2/ -03
* Release floats computation precision by enabling fast math flags.
* Not all floating point ops take the same time.

Outline

1. Floats are not real
    a. Rounding is all around
    b. Optimize like a boss (for real)
    c. Compare with care
2. Ignoring exceptions
    a. Being brave with exceptions
    b. Compiler is your friend (fortran)
    c. Compiler is (NOT) your friend (C/C++)

Ignoring exceptions

Exception	Value
Overflow	± inf
Underflow	0 or denorms
Divide by zero	± inf
Invalid	Nan
Inexact	round(x)

Ignoring exceptions

By default a result is delivered and the computation is continued.

Ignoring exceptions

This can completely mess up your results, and you will realize only when you get them

Outline

1. Floats are not real
    a. Rounding is all around
    b. Optimize like a boss (for real)
    c. Compare with care
2. Ignoring exceptions
    a. Being brave with exceptions
    b. Compiler is your friend (fortran)
    c. Compiler is (NOT) your friend (C/C++)

Being brave with exceptions

1. Say the compiler you won't ignore exceptions
2. Check for exceptions (Fortran and C/C++ have different modes)
3. Use a debugger:
   a. compile with -O0 -g
   b. Say you want core files : ulimit -c unlimited
   c. Examine core file with a debugger

Outline

1. Floats are not real
    a. Rounding is all around
    b. Optimize like a boss (for real)
    c. Compare with care
2. Ignoring exceptions
    a. Being brave with exceptions
    b. Compiler is your friend (fortran)
    c. Compiler is (NOT) your friend (C/C++)

Compiler is your friend : FORTRAN

Gnu compilation flags:

* -ffpe-trap=zero,invalid,underflow,overflow

Intel compilation flags:

* -fpe0 -fpe-all=0(for mixed c/fortran) -traceback

Outline

1. Floats are not real
    a. Rounding is all around
    b. Optimize like a boss (for real)
    c. Compare with care
2. Ignoring exceptions
    a. Being brave with exceptions
    b. Compiler is your friend (fortran)
    c. Compiler is (NOT) your friend (C/C++)

Compiler is (NOT) your friend (C/C++)

Inform the compiler you will access the floating point environment:

* GNU: -frounding-math -ftrapping-math
* INTEL: -fp-model except or fp-model strict (fp-trap-all=divzero)
* Exceptions macro are available in <fenv.h>
* You have to write code for:
  i. manually checking if a float op generates exceptions
  ii. or asking for trapping to the OS and define an handler

Checking (cpp reference is your friend)

```
#include <iostream>
#include <cfenv>
#include <cmath>

#pragma STDC FENV_ACCESS ON #don’t supported by most compilers, for intel and gnu substituted by flags listed
in the following

volatile double zero = 0.0; // volatile not needed where FENV_ACCESS is supported
volatile double one = 1.0;  // volatile not needed where FENV_ACCESS is supported

int main()
{
    std::feclearexcept(FE_ALL_EXCEPT);
    std::cout <<  "1.0/0.0 = " << 1.0 / zero << '\n';
    if(std::fetestexcept(FE_DIVBYZERO)) {
        std::cout << "division by zero reported\n";
    } else {
        std::cout << "division by zero not reported\n";
    }
[...]
}
```

TRAPPING

```
#pragma STDC FENV_ACCESS ON
#include<cstdlib>
#include<cfenv>
#include<fenv.h>
#include<signal.h>
void floating_point_handler(int signal, siginfo_t *sip, void
*uap) {
  std::cerr << "floating point error at " << sip->si_addr <<
" : ";
  int code=sip->si_code;
  if (code==FPE_FLTDIV)
    std::cerr << "division by zero\n";
  if (code==FPE_FLTUND)
    std::cerr << "underflow\n";
  if (code==FPE_FLTINV)
    std::cerr << "invalid result\n";
  std::abort();
}
```

```
int main() {
  std::feclearexcept(FE_ALL_EXCEPT);
  feenableexcept(FE_DIVBYZERO | FE_UNDERFLOW | FE_OVERFLOW |
FE_INVALID);
  struct sigaction act;
  act.sa_sigaction=floating_point_handler;
  act.sa_flags=SA_SIGINFO;
  sigaction(SIGFPE, &act, NULL);
  double zero=0.0;
  double one=1.0;
  std::cout << "1.0/1.0 = " << one/one << '\n';
  std::cout << "1.0/0.0 = " << one/zero << '\n';
  std::cout << "1.0/1.0 = " << one/one << '\n';
  return EXIT_SUCCESS;
}
```

TRAPPING

```
#pragma STDC FENV_ACCESS ON
#include<cstdlib>
#
#
#
v
*

"
  if (code==FPE_FLTUND)
    std::cerr << "underflow\n";
  if (code==FPE_FLTINV)
    std::cerr << "invalid result\n";
  std::abort();
}
```

Code from: http://numbercrunch.de/blog/2016/05/c1114-for-scientific-computing-vi/

```
int main() {
  std::feclearexcept(FE_ALL_EXCEPT);
                                                  W |

  std::cout << "1.0/1.0 = " << one/one << '\n';
  return EXIT_SUCCESS;
}
```

THANKS!

C.LATINI@CINECA.IT

BacKups

Safe math disables Unsafe Optimizations: which ones?

x / x ⇔ 1.0	x could be 0.0, ∞, or NaN
x – y ⇔ - (y – x)	If x equals y, x – y is +0.0 while – (y – x) is -0.0
x – x ⇔ 0.0	x could be ∞ or NaN
x * 0.0 ⇔ 0.0	x could be -0.0, ∞, or NaN
x + 0.0 ⇔ x	x could be -0.0
(x + y) + z ⇔ x + (y + z)	General reassociation is not value safe
(x == x) ⇔ true	x could be NaN

USEFUL LINKS(1)

1. What every computer scientist should know about floats

https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html

http://www.lahey.com/float.htm

https://docs.python.org/3/tutorial/floatingpoint.html

2. Intel compiler doc

https://software.intel.com/sites/default/files/article/326703/fp-control-2012-08.pdf

https://software.intel.com/en-us/node/581886

3. Compilers (all)

Man pages help!

USEFUL LINKS(2)

Blogs and online resources

https://possiblywrong.wordpress.com/2013/11/15/floating-point-equality-its-worse-than-you-think/

https://randomascii.wordpress.com/category/floating-point/

http://numbercrunch.de/blog/2016/05/c1114-for-scientific-computing-vi/

http://www.toves.org/books/float/

Cppreference

http://en.cppreference.com/w/cpp/numeric/fenv

C++ FAQ:

http://www.parashift.com/c++-faq-lite/floating-point-arith2.html