2-floating-point-problems

June 3, 2024

1 Flaws of floating-point computing

Floating-point numbers can represent a very large range of numbers, from the smallest to the largest, similarly to scientific notation. They are the prefered types for scientific computing. Yet, one must be aware of the many **rounding errors** which are implied.

First, in order to check visually the accuracy of some calculations, let's increase to 18 the output stream precision (this is 6 by default).

[2]:
```
#include <iostream>
std::cout.precision(18) ;
```

1.1 Binary is not decimal

The floating-point types have a limited number of digits available for their internal representation. Many numbers, such as `1./3.`, cannot be represented exactly:

[3]:
```
std::cout << (1./3.) << std::endl ;
```

```
0.333333333333333315
```

Less intuitive, some very simple numbers (for humans) do not have an exact base-two representation:

[4]:
```
std::cout << 0.1 << std::endl ;
```

```
0.100000000000000006
```

Some simple operations may amass rounding errors, which complicates comparison of floating-point numbers:

[3]:
```
double d1 = 1. ;
double d2 = .1+.1+.1+.1+.1+.1+.1+.1+.1+.1 ;

std::cout << d1 << std::endl ;
std::cout << d2 << std::endl ;

if (d1==d2)
 { std::cout<<"numbers are the same"<<std::endl ; }
else
 { std::cout<<"numbers differ !"<<std::endl ; }
```

```
1
0.999999999999999889
numbers differ !
```

[6]:
```
double d1 = 0.1+0.2 ;
double d2 = 0.3 ;
std::cout << d1 << std::endl ;
std::cout << d2 << std::endl ;
if (d1==d2)
 { std::cout<<"numbers are the same"<<std::endl ; }
else
 { std::cout<<"numbers differ !"<<std::endl ; }
```

```
0.300000000000000044
0.299999999999999989
numbers differ !
```

1.2 Good old-fashioned practice: epsilon

When comparing some floating point numbers, always allow an epsilon difference.

[22]:
```
#include <cmath>
#include <limits>
```

[20]:
```
bool compare( double val1, double val2 )
 {
  double epsilon = std::numeric_limits<double>::epsilon() ;
  return (std::abs(val1-val2)<epsilon) ;
 }
```

[21]:
```
if (compare(1.,.1+.1+.1+.1+.1+.1+.1+.1+.1+.1 ))
 { std::cout<<"numbers are the same"<<std::endl ; }
else
 { std::cout<<"numbers differ !"<<std::endl ; }
```

```
numbers are the same
```

1.3 Absorption

Adding a very small number to a very big one has no effect on the big one... And there is nothing you can do about it, except using a larger floating point type, to a given extent, and more importantly modify your algorithms so to avoid this situation. The even worse point is that it is really hard to detect such pitfall.

[3]:
```
%%file tmp.absorption.cpp

#include <iostream>
#include <stdfloat>
```

```cpp
int main( int argc, char * argv[] )
 {
  auto v1 { 128.0f16 } ;
  auto v2 { 1.f16/16 } ;
  std::cout << v1 << std::endl ;
  std::cout << v2 << std::endl ;
  std::cout << (v1+v2) << std::endl ;
 }
```

```
Overwriting tmp.absorption.cpp
```

[4]:
```
!rm -f tmp.absorption.exe && g++ -O2 -std=c++23 tmp.absorption.cpp -o tmp.
 ↪absorption.exe && ./tmp.absorption.exe
```

```
128
0.0625
128
```

1.4 Cancellation

Somehow similar to the previous problem, if you substract two numbers which are very close, the results will get very few significant digits. In the example below, where we consider the `long double` result as the "truth", after only few operations, the relative errors is far from the expected 7 significant digits.

[46]:
```cpp
%%file tmp.cancellation.cpp

#include <iostream>
#include <iomanip>

template< typename R >
std::tuple<R,R> main_impl()
 {
  R v1 { static_cast<R>(3.333) + static_cast<R>(3.0e-4) } ;
  R v2 { static_cast<R>(3.333) + static_cast<R>(2.0e-4) } ;
  R res1 = (v1*v1-v2*v2) ;
  R res2 = (v1+v2)*(v1-v2) ;
  return { res1, res2 } ;
 }

int main( int argc, char * argv[] ) {
  auto [ res1l, res2l ] = main_impl<long double>() ;
  auto [ res1f, res2f ] = main_impl<float>() ;

  std::cout << std::fixed << std::setprecision(18) ;
  std::cout << "(v1*v1-v2*v2)     float result: " << res1f << std::endl ;
  std::cout << "(v1+v2)*(v1-v2)   float result: " << res2f << std::endl ;
```

```
 std::cout << "(v1*v1-v2*v2)   relative error: " << (res1l-res1f)/res1l << std:
↪:endl ;
 std::cout << "(v1+v2)*(v1-v2) relative error: " << (res2l-res2f)/res2l << std:
↪:endl ;
}
```

```
Overwriting tmp.cancellation.cpp
```

[47]:
```
!rm -f tmp.cancellation.exe && g++ -O2 -std=c++23 tmp.cancellation.cpp -o tmp.
↪cancellation.exe
```

[48]:
```
!./tmp.cancellation.exe
```

```
(v1*v1-v2*v2)     float result: 0.000666618347167969
(v1+v2)*(v1-v2)   float result: 0.000665965897496790
(v1*v1-v2*v2)   relative error: 0.000047480435055752
(v1+v2)*(v1-v2) relative error: 0.001026179409299035
```

2 Take Away

Modern C++ will not bring any silver bullet for the rounding problems of floating point computing. You still have to rely on only some old-fashioned good practice, yet externals tools that can help to locate greatest errors (CADNA, verificarlo, verrou).

3 Questions ?