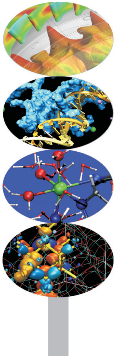
Programmazione Avanzata / 2

Francesco Salvadore

CINECA Roma - SCAI Department



Roma, 2014

Outline

Floating Point Computing

Why talking about data formats?

- The “numbers” used in computers are different from the “usual” numbers
- Some differences have known consequences
  - size limits
  - numerical stability
  - algorithm robustness
- Other differences are often misunderstood
  - portability
  - exceptions
  - surprising behaviours with arithmetic

Bits and Integers

- Computers usually handle bits
- An integer number *n* may be stored as a sequence of bits
- Of course, you have a range

$$-2^{r-1} \leq n \leq 2^{r-1} - 1$$

- Two common sizes
  - 32 bit: range $-2^{31} \leq n \leq 2^{31} - 1$
  - 64 bit: range $-2^{63} \leq n \leq 2^{63} - 1$
- Languages allow for declaring different flavours of integers
  - select the type you need compromizing on avoiding overflow and saving memory
- Is it difficult to have an integer overflow?
  - consider a cartesian discretization mesh $(1536 \times 1536 \times 1536)$ and a linearized index i

$$0 \leq i \leq 3623878656 > 2^{31} = 2147483648$$

Bits and Integers / 2

- Fortran "officially" does not let you specify the size of declared data
  - you request **`kind`** and the language do it for you
  - in principle very good, but interoperability must be considered with attention
  - and the underlying types are usually just a few of "well known" types
- C standard types do not match exact sizes, too
  - look for **`int`**, **`long int`**, **`unsigned int`**, ...
  - **`char`** is an 8 bit integer
  - unsigned integers available, doubling the maximum value $0 \leq n \leq 2^r - 1$

Bits and Reals

- Note: From now on, some examples will consider base 10 numbers just for readability
- Representing reals using bits is not natural
- Fixed size approach
  - select a fixed point corresponding to comma
  - e.g., with 8 digits and 5 decimal places 36126234 gets interpreted as 361.26234
- Cons:
  - limited range: from 0.00001 to 999.99999, spanning 10^8
  - only numbers having at most 5 decimal places can be exactly represented
- Pros:
  - constant resolution, i.e. the distance from one point to the closest one (0.00001)

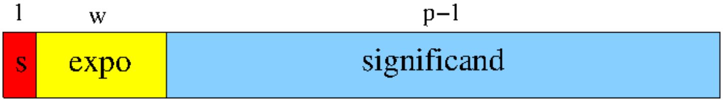
Floating point approach

- Consider scientific notation

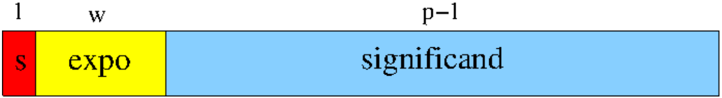
$$n = (-1)^s \cdot m \cdot \beta^e$$

$$0.0046367 = (-1)^0 \cdot 4.6367 \cdot 10^{-3}$$

- Represent it using bits
  - one digit for sign *s*
  - "p-1" digits for significand (mantissa) *m*
  - "w" digits for exponent *e*



Exponent and Significand



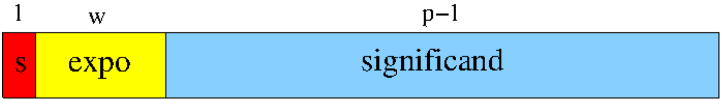
- Exponent
  - unsigned biased exponent
  - $e_{min} \leq e \leq e_{max}$
  - e_{min} must be equal to $(1 - e_{max})$
- Mantissa
  - precision p, the digits x_i are $0 \leq x_i < \beta$

$$m = \sum_{i=0}^{p-1} x_i \cdot \beta^{-i}$$

  - “hidden bit” format used for normal values: 1.xx...x

IEEE Name	Format	Storage Size	w	p	e_{min}	e_{max}
Binary32	Single	32	8	24	-126	+127
Binary64	Double	64	11	53	-1022	+1023
Binary128	Quad	128	15	113	-16382	+16383

Mantissa



- Cons:
  - only “some” real numbers are floating point numbers (see later)
- Pros:
  - constant relative resolution (relative precision), each number is represented with the same *relative error* which is the distance from one point to the closest one divided by the number (see later)
  - wide range: “normal” positive numbers from $10^{e_{min}}$ to $9,999..9 \cdot 10^{e_{max}}$
- The representation is unique assuming the mantissa is

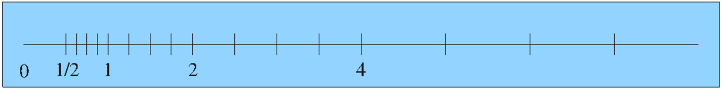
$$1 \leq m < \beta$$

i.e. using “normal” floating-point numbers

Resolution

- The distance among “normal” numbers is not constant
- E.g., $\beta = 2$, $p = 3$, $e_{min} = -1$ and $e_{max} = 2$:
  - 16 positive “normalized” floating-point numbers

```
e = -1 ; m = 1 + [0:1/4:2/4:3/4] ==> [4/8:5/8:6/8:7/8]
e =  0 ; m = 1 + [0:1/4:2/4:3/4] ==> [4/4:5/4:6/4:7/4]
e = +1 ; m = 1 + [0:1/4:2/4:3/4] ==> [4/2:5/2:6/2:7/2]
e = +2 ; m = 1 + [0:1/4:2/4:3/4] ==> [4/1:5/1:6/1:7/1]
```



SCAI SuperComputing Applications and Innovation

Relative Resolution

SAPIENZA Università di Roma

- What does it mean "constant relative resolution"?
- Given a number $N = m \cdot \beta^e$ the nearest number has distance

$$R = \beta^{-(p-1)}\beta^e$$

  - E.g., given $3.536 \cdot 10^{-6}$, the nearest (larger) number is $3.537 \cdot 10^{-6}$ having distance $0.001 \cdot 10^{-6}$
- The relative resolution is (nearly) constant (considering $1 \leq m < \beta$)

$$\beta^{-p} < \frac{R}{N} = \frac{\beta^{-(p-1)}}{m} \leq \beta^{-(p-1)}$$

Intrinsic Error

- Not any real number can be expressed as a floating point number
  - because you would need a larger exponent
  - or because you would need a larger precision
- The resolution is directly related to the intrinsic error
  - if $p = 4$, 3.472 may approximate numbers between 3.4715 and 3.4725, its intrinsic error is 0.0005
  - the instrinsic error is (less than) $(\beta/2)\beta^{-p}\beta^{e}$
  - the relative intrinsic error is

$$(1/2)\beta^{-p} < \frac{(\beta/2)\beta^{-p}}{m} \leq (\beta/2)\beta^{-p} = \varepsilon$$

  - The intrinsic error ε is also called “machine epsilon” or “relative precision”

Measuring error

- When performing calculations, floating-point error may propagate and exceed the intrinsic error

```
real value              = 3.14145
correctly rounded value = 3.14
current value           = 3.17
```

- The most natural way to measure rounding error is in “ulps”, i.e. units in the last place
  - e.g., the error is 3 ulps
- Another interesting possibility is using “machine epsilon”, which is the relative error corresponding to 0.5 ulps

```
error   = 3.17-3.14145 = 0.02855
machine epsilon  = 10/2*0.001   = 0.005
relative error   = 5.71 ε
```

Handling errors

- Featuring a constant relative precision is very useful when dealing with rescaled equations
- Beware:
  - 0.1 has just one decimal digit using radix 10, but is periodic using radix 2
- the exact binary representation would have a "1100" sequence continuing endlessly:
  $e = -4; s = 110011001100110011001100110011001 \ldots$
- When rounded to 24 bits this becomes
  $e = -4; s = 110011001100110011001101$, which is actually 0.100000001490116119384765625 in decimal.
- periodicity arises when the fractional part has prime factors not belonging to the radix
- by the way, in Fortran if `a` is double precision, `a=0.2` is badly approximated (use `a=0.2d0` instead)
- Beware overflow!
  - you think it will not happen with your code but it may happen
  - exponent range is symmetric: if possibile, perform calculations around 1 is a good idea

Types features

IEEE Name	min	max	ε	C	Fortran
Binary32	1.2E-38	3.4E38	5.96E-8	float	real
Binary64	2.2E-308	1.8E308	1.11E-16	double	real(kind(1.d0))
Binary128	3.4E-4932	1.2E4932	9.63E-35	long double	real(kind=...)

- There are also “double extended” type and parametrized types
- Extended and quadruple precision devised to limit the round-off during the double calculation of trascendental functions and increase overflow
- Extended and quad support depends on architecture and compiler: often emulated and, hence, slow!
- Decimal with 32, 64 and 128 bits are defined by standards, too
- FPU are usually “conformant” but not “compliant”
- To be safe when converting binary to text specify 9 decimals for single precision and 17 decimal for double

Error propagation

- Assume $p = 3$ and you have to compute the difference $1.01 \cdot 10^1 - 9.93 \cdot 10^0$
- To perform the subtraction, usually a shift of the smallest number is performed to have the same exponent
- First idea: compute the difference exactly and then round it to the nearest floating-point number

$$x = 1.01 \cdot 10^1 \quad ; \quad y = 0.993 \cdot 10^1$$

$$x - y = 0.017 \cdot 10^1 = 1.70 \cdot 10^{-2}$$

- Second idea: compute the difference with $\mathbf{p}$ digits

$$x = 1.01 \cdot 10^1 \quad ; \quad y = 0.99 \cdot 10^1$$

$$x - y = 0.02 \cdot 10^1 = 2.00 \cdot 10^{-2}$$

the error is 30 ulps!

Guard digit

- A possibile solution: use the guard digit (p+1 digits)

$$x = 1.010 \cdot 10^1$$

$$y = 0.993 \cdot 10^1$$

$$x - y = 0.017 \cdot 10^1 = 1.70 \cdot 10^{-2}$$

- Theorem: if x and y are floating-point numbers in a format with parameters β and p, and if subtraction is done with p + 1 digits (i.e. one guard digit), then the relative rounding error in the result is less than 2 ε.

Cancellation

- When subtracting nearby quantities, the most significant digits in the operands match and cancel each other
- There are two kinds of cancellation: catastrophic and benign
  - benign cancellation occurs when subtracting exactly known quantities: according to the previous theorem, if the guard digit is used, a very small error results
  - catastrophic cancellation occurs when the operands are subject to rounding errors
- For example, consider $b = 3.34$, $a = 1.22$, and $c = 2.28$.
  - the exact value of $b^2 - 4ac$ is 0.0292
  - but b^2 rounds to 11.2 and $4ac$ rounds to 11.1, hence the final answer is 0.1 which is an error by $70 ulps$
  - the subtraction did not introduce any error, but rather exposed the error introduced in the earlier multiplications.

Cancellation / 2

- The expression $x^2 - y^2$ is more accurate when rewritten as $(x - y)(x + y)$ because a catastrophic cancellation is replaced with a benign one
  - replacing a catastrophic cancellation by a benign one may be not worthwhile if the expense is large, because the input is often an approximation
- Eliminating a cancellation entirely may be worthwhile even if the data are not exact
- Consider second-degree equations

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

  - if $b^2 >> ac$ then $b^2 - 4ac$ does not involve a cancellation
  - but, if $b > 0$ the addition in the formula will have a catastrophic cancellation.
  - to avoid this, multiply the numerator and denominator of x_1 by $-b - \sqrt{b^2 - 4ac}$ to obtain $x_1 = (2c)/(-b - \sqrt{b^2 - 4ac})$ where no catastrophic cancellation occurs

Rounding and IEEE standards

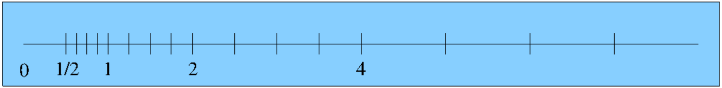
- The IEEE standards requires correct rounding for:
  - addition, subtraction, mutiplication, division, remainder, square root
  - conversions to/from integer
- The IEEE standards recommends correct rounding for:
  - e^x, $e^x - 1$, 2^x, $2^x - 1$, $\log_\alpha(\phi)$, $1/\sqrt(x)$, $sin(x)$, $cos(x)$, $tan(x)$,....
- Remember: “No general way exists to predict how many extra digits will have to be carried to compute a transcendental expression and round it correctly to some preassigned number of digits” (W. Kahan)

Special values

- Zero: signed
- Infinity: signed
  - overflow, divide by 0
  - Inf-Inf, Inf/Inf, 0 $\cdot$ Inf $\rightarrow$ NaN (indeterminate)
  - Inf op a $\rightarrow$ Inf if a is finite
  - a / Inf $\rightarrow$ 0 if a is finite
- NaN: not a number!
  - Quiet NaN or Signaling NaN
  - e.g. $\sqrt{a}$ with $a < 0$
  - NaN op a $\rightarrow$ NaN or exception
  - NaNs do not have a sign: they aren't a number
  - The sign bit is ignored
  - NanS can "carry" information

Zero and Denormals

- Considering positve numbers, the smallest "normal" floating point number is $n_{smallest} = 1.0 \cdot \beta^{e_{min}}$
- In the previous example it is $1/2$



- At least we need to add the zero value
  - there are two zeros: **+0** and **−0**
- When a computation result is less than the minimum value, it could be rounded to zero or to the minimum value

Zero and Denormals / 2

- Another possibility is to use denormal (also called subnormal) numbers
  - decreasing mantissa below 1 allows to decrease the floating point number, e.g. $0.99 \cdot \beta^{e_{min}}$, $0.98 \cdot \beta^{e_{min}}, \ldots, 0.01 \cdot \beta^{e_{min}}$
  - subnormals are linearly spaced and allow for the so called "gradual underflow"
- Pro: $k/(a-b)$ may be safe (depending on k) even is $a-b < 1.0 \cdot \beta^{e_{min}}$
- Con: performance of denormals are significantly reduced (dramatic if handled only by software)
- Some compilers allow for disabling denormals
  - Intel compiler has **`-ftz`**: denormal results are flushed to zero
  - automatically activated when using any level of optimization!

Walking Through

- Double precision: w=11 ; p=53

```
0x0000000000000000  +zero
0x0000000000000001  smallest subnormal
...
0x000fffffffffffff  largest subnormal
0x0010000000000000  smallest normal
...
0x001fffffffffffff
0x0020000000000000  2 X smallest normal
...
0x7fefffffffffffff  largest normal
0x7ff0000000000000  +infinity
```

Walking Through

```
0x7ff0000000000001  NaN
...
0x7fffffffffffffff  NaN
0x8000000000000000  -zero
0x8000000000000001  negative subnormal
...
0x800fffffffffffff  'largest' negative subnormal
0x8010000000000000  'smallest' negative normal
...
0xfff0000000000000  -infinity
0xfff0000000000001  NaN
...
0xffffffffffffffff  NaN
```

Error-Free Transformations

- An error-free transformation (EFT) is an algorithm which determines the rounding error associated with a floating-point operation
- E.g., addition/subtraction

$$a + b = (a \oplus b) + t$$

  where $\oplus$ is a symbol for floating-point addition
- Under most conditions, the rounding error is itself a floating-point number
- **An EFT can be implemented using only floating-point computations in the working precision**

EFT for Addition

- FastTwoSum: compute $a + b = s + t$ where

$$|a| \geq |b|$$

$$s = a \oplus b$$

```
void FastTwoSum( const double a, const double b,
                 double* s, double* t ) {
            // No unsafe optimizations !
            *s = a + b;
            *t = b - ( *s - a );
            return;
      }
```

EFT for Addition / 2

- No requirements on $|a|$ or $|b|$
- Beware: avoid compiler unsafe optimizations!

```
void TwoSum( const double a, const double b,
                 double* s, double* t ) {
            // No unsafe optimizations !
            *s = a + b;
            double z = *s - b;
            *t = ( a-z )+( b-( *s-z ) );
            return;
      }
```

Summation techniques

- Condition number

$$C_{sum} = \frac{\sum |a_i|}{|\sum a_i|}$$

- If C_{sum} is " not too large", the problem is not ill conditioned and traditional methods may suffice
- But if it is "too large", we want results appropriate to higher precision without actually using a higher precision
- But if higher precision is available, consider to use it!
  - beware: quadruple precision is nowadays only emulated

Traditional summation

$$s = \sum_{i=0}^{n} x_i$$

```
double Sum( const double* x, const int n ) {
    int i;
    for ( i = 0; i < n; i++ ) {
        Sum += x[ i ];
    }
    return Sum;
}
```

- Traditional Summation: what can go wrong?
  - catastrophic cancellation
    - magnitude of operands nearly equal but signs differ
    - loss of significance
  - small terms encountered when running sum is large
    - the smaller terms don't affect the result
    - but later large magnitude terms may reduce the running sum

Sorting and Insertion

- Reorder the operands; sort by
  - Increasing magnitude value
- Insertion
  - First sort by magnitude
  - Remove the first two item and compute their sum
  - Insert the result on the list, keeping list sorted
  - Repeat until only one element is left on the list
- Many variations

Kahan summation

- Based on FastTwoSum and TwoSum techniques
- Knowledge of the exact rounding error in a floating-point addition is used to correct the summation
- Compensated Summation

```
double Kahan( const double* a, const int n ) {
   double s = a[ 0 ];          //  sum
   double t = 0.0;             //  correction term
   for(int i=1; i<n ; i++) {
       double y = a[ i ] - t;  //  next term "plus" correction
       double z = s + y;       //  add to accumulated sum
       t = ( z - s ) - y;      //  t <- -( low part of y )
       s = z;                  //  update sum
   }
   return s;
}
```

More

- Many variations known (Knutht, Priest,...)
- Sort the values and sum starting from smallest values (for positive numbers)
- Other techniques (distillation)
- Use a greater precision or emulate it (long accumulators)
- Similar problems for Dot Product, Polynomial evaluation,...

Exceptions (IEEE 754-2008)

- Underflow
  - Absolute value of a non zero result is less than the minimum value (i.e., it is subnormal or zero)
- Overflow
  - Magnitude of a result greater than the largest finite value
  - Result is $\pm\infty$
- Division by zero
  - a/b where a is finite and non zero and b=0
- Inexact
  - Result, after rounding, is not exact
- Invalid
  - an operand is sNaN, square root of negative number or combination of infinity

Handling exceptions

- Let us say you may produce a NaN
- What do you want to do in this case?

- First scenario: go on, there is no error and my algorithm is robust
- E.g., the function **`maxfunc`** compute the maximum value of a scalar function $f(x)$ testing each function value corresponding to the grid points **`g(i)`**

```
        call maxfunc(f,g)
```

  - to be safe I should pass the domain of f but it could be difficult to do
  - I may prefer to check each grid point **`g(i)`**
  - if the function is not defined somewhere, I will get a NaN (or other exception) but I do not care: the maximum value will be correct

Handling exceptions / 2

- Second scenario: ops, something went wrong during the computation...
- (Bad) solution: complete your run and check the results and, if you see NaN, throw it away
- (First) solution: trap exceptions using compiler options (usually systems ignore exception as default)
- Some compilers allow to enable or disable floating point exceptions
  - Intel compiler: **`-fpe0`**: Floating-point invalid, divide-by-zero, and overflow exceptions are enabled. If any such exceptions occur, execution is aborted.
  - GNU compiler: **`-ffpe-trap=zero,overflow,invalid,underflow`**
- very useful, but the performance loss may be material!
- use only in debugging, not in production stage

Handling exceptions / 3

- (Second) solution: check selectively
  - each N_{check} time-steps
  - the most dangerous code sections
- Using language features to check exceptions or directly special values (NaNs,...)
  - the old print!
  - Fortran (2003): from module **`ieee_arithmetic`**, **`ieee_is_nan(x)`**, **`ieee_is_finite(x)`**
  - C: from **`<math.h>`**, **`isnan`** or **`isfinite`**, from C99 look for **`fenv.h`**
  - do not use old style checks (compiler may remove them):

```
int IsFiniteNumber(double x) {
    return (x <= DBL_MAX && x >= -DBL_MAX);
}
```

Floating-point control

- Why doesn't my application always give the same answer?
  - inherent floating-point uncertainty
  - we may need reproducibility (porting, optimizing,...)
  - accuracy, reproducibility and performance usually conflict!
- Compiler safe mode: transformations that could affect the result are prohibited, e.g.
  - $x/x = 1.0$, false if $x = 0.0, \infty, NaN$
  - $x - y = -(y - x)$ false if $x = y$, zero is signed!
  - $x - x = 0.0$ ...
  - $x * 0.0 = 0.0$ ...

Floating-point control / 2

- An important case: reassociation is not safe with floating-point numbers
  - $(x + y) + z = x + (y + z)$: reassociation is not safe
  - compare

$$-1.0+1.0e-13+1.0 \; = \; 1.0-1.0+1.0e-13 \; = \; 1.0e-13+1.0-1.0$$

  - $a * b/c$ may give overflow while $a * (b/c)$ does not
- Best practice:
  - select the best expression form
  - promote operands to the higher precision (operands, not results)

Floating-point control / 3

- Compilers allow to choose the safety of floating point semantics
- GNU options (high-level):

```
-f[no-]fast-math
```

  - It is off by default (different from icc)
  - Also sets abrupt/gradual underflow (FTZ)
  - Components control similar features, e.g. value safety (`-funsafe-math-optimizations`)
- For more detail
  `http://gcc.gnu.org/wiki/FloatingPointMath`

Floating-point control / 4

- Intel options:

```
-fp-model <type>
```

  - fast=1: allows value-unsafe optimizations (**default**)
  - fast=2: allows additional approximations
  - precise: value-safe optimizations only
  - strict: precise + except + disable fma

- Also pragmas in C99 standard

```
#pragma STDC FENV_ACCESS etc
```

Endianness

- Which is the ordering of bytes in memory? E.g.,

```
-1267006353 ===> 10110100011110110000010001101111
```

  - Big endian: `10110100 01111011 00000100 01101111`
  - Little endian: `01101111 00000100 01111011 10110100`
  - Other exotic layouts (VAX,...) nowadays unusual
  - Limits portability
- Possibile solutions
  - conversion binary to text and text to binary
  - compiler extensions(Fortran):
    - HP Alpha, Intel: -convert big_endian | little_endian
    - PGI: -byteswapio
    - Intel, NEC: F_UFMTENDIAN (variabile di ambiente)
  - explicit reoredering
  - conversion libraries

C and Fortran data portability

- For C Standard Library a file is written as a stream of byte
- In Fortran file is a sequence of records:
  - each read/write refer to a record
  - there is record marker before and after a record (32 or 64 bit depending on file system)
  - remember also the different array layout from C and Fortran
- Possible portability solutions:
  - read Fortran records from C
  - perform the whole I/O in the same language (usually C)
  - use Fortran 2003 **`access='stream'`**
  - use I/O libraries

Which precision do I need?

- Single, Double or Quad?
  - maybe single is too much!
  - computations get (much) slower when increasing precision, storage increases and power supply too
- Famous story
  - Patriot missile incident (2/25/91) . Failed to stop a scud missile from hitting a barracks, killing 28
  - System counted time in 1/10 sec increments which doesn't have an exact binary representation. Over time, error accumulates.
  - The incident occurred after 100 hours of operation at which point the accumulated errors in time variable resulted in a 600+ meter tracking error.
- **Wider floating point formats turn compute bound problems into memory bound problems!**

Which precision do I need?/2

- Programmers should conduct mathematically rigorous analysis of their floating point intensive applications to validate their correctness
- Training of modern programmers often ignores numerical analysis
- Useful tricks
  - Repeat the computation with arithmetic of increasing precision, increasing it until a desired number of digits in the results agree
  - Repeat the computation in arithmetic of the same precision but rounded differently, say Down then Up and perhaps Towards Zero, then compare results
  - Repeat computation a few times in arithmetic of the same precision but with slightly different input data, and see how widely results vary

Interval arithmetic

- A “correct” approach
- Interval number: possible values within a closed set

$$\mathbf{x} \equiv [x_L, x_R] := \{x \in \mathbb{R} | x_L \leq x \leq x_R\}$$

  - e.g., 1/3=0.33333 ; 1/3 $\in$ [0.3333,0.3334]
- Operations: let $x = [a, b]$ and $y = [c, d]$
  - Addition x + y = [a, b] + [c, d] = [a + c, b + d]
  - Subtraction x - y = [a, b] - [c, d] = [a -d, b -c]
  - ...
- Properties are interesting and can be applied to equations
- Interval Arithmetic has been tried for decades, but often produces bounds too loose to be useful
- A possible future
  - chips supporting variable precision and uncertainty tracking
  - runs software at low precision, tracks accuracy and reruns computations automatically if the error grows too large.

References

- N.J. Higham, Accuracy and Stability of Numerical Algorithms 2nd ed., SIAM, capitoli 1 e 2
- D. Goldberg, What Every Computer Scientist Should Know About Floating-Point Arithmetic, ACM C.S., vol. 23, 1, March 1991 http://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html
- W. Kahan http://www.cs.berkeley.edu/ wkahan/
- Standards: http://grouper.ieee.org/groups/754/

Hands-on: Compensated sum

- The code in **`summation.cpp/f90`** initializes an array with an ill-conditioned sequence of the order of

```
100,-0.001,-100,0.001,.....
```

- Simple and higher precision summation functions are implemented
- Implement Kahan algorithm in C++ or Fortran
- Compare the accuracy of the results

Hands-on: C++ Solution


```
REAL_TYPE summation_kahan( const REAL_TYPE a[],
                           const size_t n_values )
{
  REAL_TYPE s = a[ 0 ];          //  sum
  REAL_TYPE t = 0;               //  correction term
  for( int i = 1; i < n_values; i++ ) {
      REAL_TYPE y = a[ i ] - t;  //  next term "plus" correction
      REAL_TYPE z = s + y;       //  add to accumulated sum
      t = ( z - s ) - y;         //  t <- -( low part of y )
      s = z;                     //  update sum
  }
  return s;
}
```

```
Summation simple  :   35404.96093750000000000
Summation Kahan   :   35402.85156250000000000
Summation higher  :   35402.85546875000000000
```

Hands-on: Fortran Solution


```
function sum_kahan(a,n)
    integer :: n
    real(my_kind) :: a(n)
    real(my_kind) :: s,t,y,z

    s=a(1)                 ! sum
    t=0._my_kind           ! correction term
    do i=2,n
       y = a(i) - t        ! next term "plus" correction
       z = s + y           ! add to accumulated sum
       t = (z-s) - y       ! t <- -( low part of y )
       s = z               ! update sum
    enddo
    sum_kahan = s
end function sum_kahan
```

```
Summation simple:   -13951.87109375000000
Summation Kahan:    -13951.91113281250000
Summation Higher:   -13951.91210937500000
```

Rights & Credits