

Floating-point arithmetic and error analysis (AFAE)

# Introduction to computer arithmetic

**Stef Graillat**

LIP6/PEQUAN – Sorbonne University

Lecture Master 2 SFPN – MAIN5



# Outline

- 1 Introduction
- 2 Number representations
- 3 Representing numbers to compute with them
- 4 Floating-point arithmetic
- 5 Use of Computer Arithmetic: Historical Failures
- 6 Conclusion

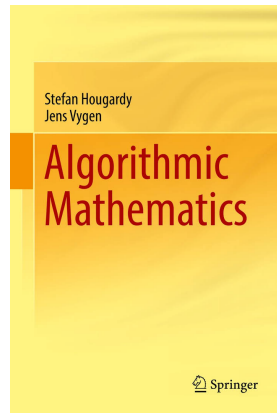
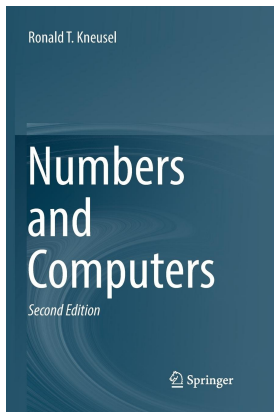
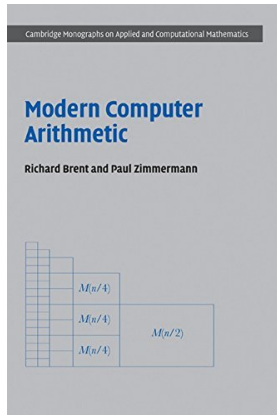
# Outline

- 1 Introduction
- 2 Number representations
- 3 Representing numbers to compute with them
- 4 Floating-point arithmetic
- 5 Use of Computer Arithmetic: Historical Failures
- 6 Conclusion

# Computer arithmetic

- Computer Arithmetic is about
  - how to represent **Numbers in a Computer**
  - Implementation of **basic operations** like  $+$ ,  $-$ ,  $\times$ ,  $/$ 
    - with nothing but bare transistors
    - with other high-level operations in software
  - Approximation and **Error Analysis**
    - Floating-Point arithmetic
    - Compute less precisely to go faster
  - Base **Algorithms for computation-intensive problems**
    - Cryptography
    - Scientific Computing
- Computer Arithmetic is **not** about but **very close** to
  - Computing with Formulas  $\Rightarrow$  Computer Algebra
  - Modelization and Problem Solving  $\Rightarrow$  Numerical Analysis

# Bibliography



# AFAE Master 2 course – organization

- Planning for the AFAE Course

Date	Subject	Organization	MAIN5	Teacher
06/10/20	Introduction to computer arithmetic	lect + tuto	✓	S. Graillat
13/10/20	Introduction to computer arithmetic (cont'd)	tuto	✓	V. Ménessier-Morain
20/10/20	Increasing the accuracy, examples with polynomials	lect + lab	✓	S. Graillat
27/10/20	Large scale rounding error analysis, conditioning	lect + tuto	✓	S. Graillat
03/11/20	Multiprecision and Interval Arithmetic	lect + lab	✓	S. Graillat
17/11/20	Stochastic Arithmetic	lect + lab	✓	F. Jézéquel
24/11/20	Validation of linear systems	lect	✓	S. Graillat
08/12/20	Fixed-Point Arithmetic	lect + tuto		Th. Hilaire
15/12/20	Elementary Functions	lect		S. Graillat
05/01/21	Faithfully rounded algorithms	lect		S. Graillat
12/01/21	Probabilistic rounding error analysis	lect + tuto		Th. Mary
19/01/21	Arbitrary precision arithmetic	tuto + lab		M. Mezzarobba
26/01/21	Scientific Articles (Group 1)	talks		You!
02/02/21	Scientific Articles (Group 2)	talks		You!

- URL: <https://www-pequan.lip6.fr/~graillat/teach/afae/>  
Login: afae2020 – Password: 2020AFAE

# AFAE Master 2 Course – evaluation (SFPN)

## Evaluation. SFPN

- 40% of the grade for your abstract and presentation
  - Jan. 26th and Feb. 2nd
  - Approx. 20 min presentation of a scientific article
  - Approx. 15 min of questions on the article and the course
  - Article list will be given Nov. 24th, 2020
  - 2 page abstract to be written by Jan. 19th, 2021.
  - Presentation skills are really important!
- 30% of the grade for the final exam,
  - In Feb. 2021
  - Several exercises covering all parts of the course
- 30% of the grade for practical work
  - Tutorial on 20/10/20, 03/11/20, 24/11/20 will be graded
  - All functions to be programmed out, tested and tuned

## Evaluation. MAIN

- 50% of the grade for **practical work**
  - Tutorial on 20/10/20, 03/11/20, 24/11/20 will be graded
  - All functions to be programmed out, tested and tuned
- 50% of the grade for the **final exam**,
  - Same examination sheet as for the SFPN people
  - The MAIN people do not give answers for certain exercises.
  - The grading scheme is different for MAIN and SFPN.



# Do you speak english?

- This is a **research-oriented Master 2 Course**
- The scientific language is **English**
  - Most of the slide-sets will be in English
  - The scientific articles you will present, too
- The Course will be read in **French**
  - **Ask questions** for what you don't understand on the slides
  - Start a small specialized vocabulary list right away
    - the carry - la retenue
    - the radix - la base
    - ...

# Outline

- 1 Introduction
- 2 Number representations**
- 3 Representing numbers to compute with them
- 4 Floating-point arithmetic
- 5 Use of Computer Arithmetic: Historical Failures
- 6 Conclusion

# Early number representations

- Neolithic: invention of the unary representation of integers
- Antiquity; invention of

alphabetic systems  
(Rome, Egypt, Greece, China)



position systems  
(Babylonia, India, Mayan)



# Position systems

Our decimal system is an example of a position system:

- The position  $i$  of a digit gives its weight  $10^i$
- Example:  $789 = 7 \cdot 10^2 + 8 \cdot 10^1 + 9 \cdot 10^0$
- With 3 decimal digits we represent from 000 to  $999 = 10^3 - 1$
- In general, using  $n$  digits, we can represent integers in  $[0..10^n - 1]$

## First advantage of position systems

With 10 symbols only, we can represent arbitrarily large numbers...

# Position system in radix $\beta$

- For any  $\beta$  integer  $\geq 2$ ,
- If we have  $\beta$  symbols (which is called  $\beta$ -ary digits)
- with  $n$  such digits, we represent the interval

$$[0.. \beta^n - 1]$$

- $d_{n-1}d_{n-2}...d_1d_0$  represents  $\sum_{i=0}^{n-1} d_i \beta^i$
- Example:  $789 = 7 \cdot 10^2 + 8 \cdot 10^1 + 9 \cdot 10^0$
- $\beta^i$  is the weight of digit  $d_i$ .

**Babylonian system is a position system in base 60.**

Why 60? And why does our hour have 60 minutes?




























































# More on the Babylonian system

- Radix 60 needs to represent 60 different digits... 60 symbols?
- Babylonians represented digits using an alphabetic system with two symbols only:  $\text{T}$  for 1, and  $\text{<}$  for 10



YBC7289 from <http://www.math.ubc.ca/~cass/euclid/ycb/ycb.html>

# Babylonian digits

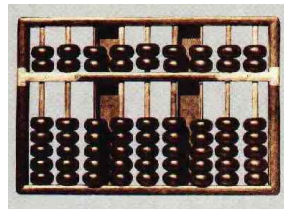
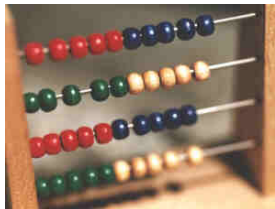
 1	 11	 21	 31	 41	 51
 2	 12	 22	 32	 42	 52
 3	 13	 23	 33	 43	 53
 4	 14	 24	 34	 44	 54
 5	 15	 25	 35	 45	 55
 6	 16	 26	 36	 46	 56
 7	 17	 27	 37	 47	 57
 8	 18	 28	 38	 48	 58
 9	 19	 29	 39	 49	 59
 10	 20	 30	 40	 50	

Taken from [http://en.wikipedia.org/wiki/Babylonian\\_numerals](http://en.wikipedia.org/wiki/Babylonian_numerals)

# Taking into account the limitations of the technology



- Our brain is able to distinguish in a glance 3 pebbles from 4, but not 7 pebbles from 8...
- On the other hand, it is good at recognizing shapes
- For this reason, Babylonians fixed the shape of the digits





# Taking into account the limitations of the technology



Digital electronic technology is based on 2-value (Boolean) logic.

## Positive integers in the binary position system

- Radix  $\beta = 2$
- Two digits 0 and 1
- Bit for “binary digit”
- With  $n$  bits we may represent numbers in  $[0..2^n - 1]$

# Fancy radix $\beta$ representations

- So far our digit set was  $\{0, \beta - 1\}$
- But any digit set including zero is OK.

## The Setun ternary computer

- Built in 1958 by Moscow State University
- Radix 3, digit set  $\{-1, 0, 1\}$  instead of  $\{0, 1, 2\}$
- example:  $1111 = 27 + 9 + 3 + 1 = 40$
- example:  $\overline{1111} = -27 - 9 - 3 - 1 = -40$
- $n$  such trits provide  $[-(3^n - 1)/2 \dots (3^n - 1)/2]$
- As many integers ( $3^n$ ) as with the standard digit set

$\overline{111}$	...	$\overline{0001}$	$\overline{0000}$	$\overline{0001}$	$\overline{0011}$	$\overline{0010}$	$\overline{0011}$	...	$\overline{1111}$
-40	...	-1	0	1	2	3	4	...	40

Other nice properties (rounding by truncation) but...  
so far, electronics are binary (+5V vs. 0V).

# To probe further on fancy radix $\beta$ representations

- So far our digit set was  $n$  digits including zero.
- We may use more than  $\beta$  digits...
- The system then becomes redundant
  - more than one way to represent some numbers
  - example: radix 10, digits  $\overline{7}, \overline{6} \dots 0, 1, \dots, 7$ ,  
 $17 = 2\overline{3}$

Studied by Cauchy (1840) for digits  $-5 \dots 5$ ,  
then Algirdas A. Avižienis in the 1960s.

If the FP unit in your cellphone includes a division unit, it probably uses Avižienis' ideas.

# Stacking several number representations

A very general technique:

- Alphabetic systems is built on top of the unary representation
- Babylonian is a position system with digits represented in an alphabetic system
- Same for Chinese abacus
- Your calculators use a decimal representation where each digit is represented in binary
- GMP multiple-precision library uses a radix- $2^{32}$  system where each digit is a 32-bit integer represented in radix 2
- ...

use at each level the representation that is adequate to this level

# What is a good number representation system?

It is a system that

- 1 is well matched to the technology available
- 2 enables “good” representation of numbers
- 3 even more important, enables “good” algorithms for the arithmetic operations

Try to design the multiplication algorithm for Roman numerals...

## Very few basic techniques of number representations

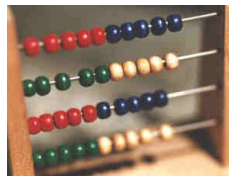
- but many parameters each
- and they can be stacked on top of each other

# Outline

- 1 Introduction
- 2 Number representations
- 3 Representing numbers to compute with them**
- 4 Floating-point arithmetic
- 5 Use of Computer Arithmetic: Historical Failures
- 6 Conclusion

# Back to the history

- Neolithic: invention of a non-volatile memory able to store positive integers represented in unary.
- Addition in unary: pour one bag into the other.
- Far antiquity: invention of the pocket calculator in China or in Babylonia
  - still essentially a memory device
  - The real invention is the position system!



# Position systems are great for addition

$$\begin{aligned} X + Y &= \sum_{i=0}^{n-1} x_i \beta^i + \sum_{i=0}^{n-1} y_i \beta^i \\ &= \sum_{i=0}^{n-1} (x_i + y_i) \beta^i \end{aligned}$$

Problem:  $x_i + y_i$  sometimes won't be a digit (e.g.  $9+9=18$ )

## carry propagation

- Remove  $\beta$  from the digit sum
- Add 1 to the digit sum to the left



# A bit of mathematical obfuscation

Let TA be the radix  $\beta$  addition table:

$$\text{TA} : \{0, 1\} \times \{0..\beta - 1\} \times \{0..\beta - 1\} \rightarrow \{0..1\} \times \{0..\beta - 1\}$$

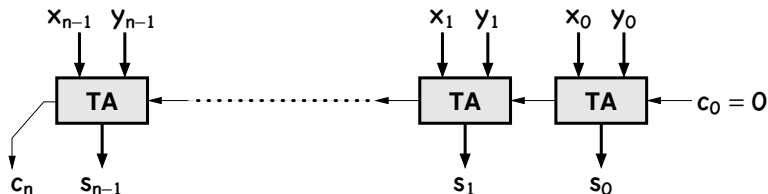
$$\begin{aligned} \text{TA}(c, x, y) &= (c', s) \\ &\text{such that } \beta c' + s = c + x + y \end{aligned}$$

- $c$  is an input carry. Assume its value is 0 or 1.
- $x$  and  $y$  are two radix- $\beta$  digits
- Their sum is between 0 and  $2\beta - 1$ . Therefore it can be written in radix  $\beta$  as two digits  $c's$ .
- The higher weight digit is called the output carry, and its value is 0 or 1 whatever the radix.

# Integer addition algorithm

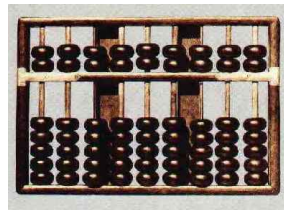
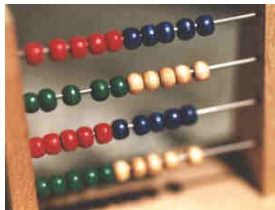
$X$  and  $Y$  two numbers written in radix  $\beta$  on  $n$  digits:

- 1:  $c_0 = 0$
- 2: **for**  $i = 0$  to  $n - 1$  **do**
- 3:    $(c_{i+1}, s_i) = \text{TA}(c_i, x_i, y_i)$
- 4: **end for**



GMP addition is exactly this loop, using as TA the `adc` processor instruction.

# The early mechanical computer



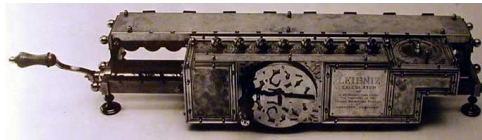
- Obviously radix 10
- Digit addition by unary addition of beads
- Carry propagation performed by the hands of the operator

# DaVinci / Schickard / Pascal machines

- 1500: Leonardo Da Vinci draws (without building it) a calculating machine
- 1623: Wilhelm Schickard reports building one
- 1645: Blaise Pascal builds 50 Pascaline (sells 15)



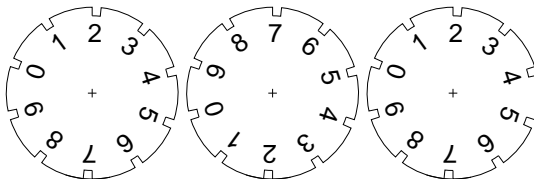
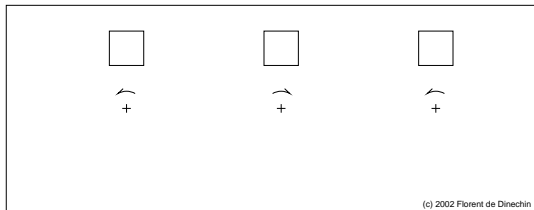
- 1679: Leibniz improves it to perform multiplication and division



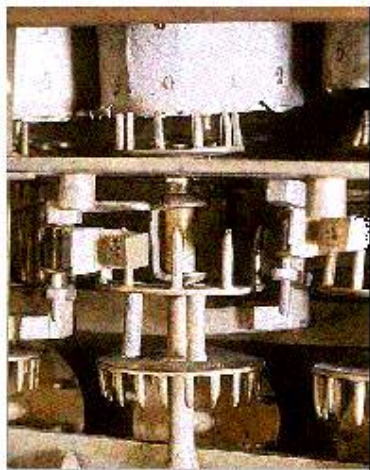
# DIY version in overhead foils

- rotating wheels perform modulo 10 addition
- carry propagation from one wheel to the next

La machine de Leibniz/Pascal/Schickard/da Vinci/...

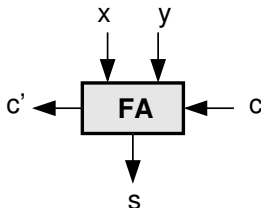


# The TA function in Pascal's machine

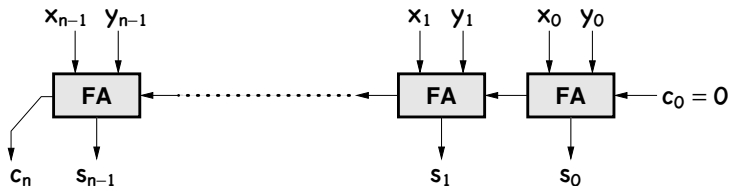


# And it even works in binary

TA is now called Full Adder:



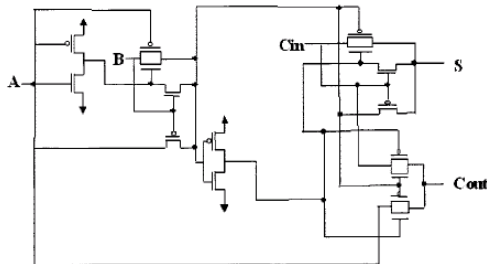
(the three inputs now have interchangeable roles)



This cheap binary adder is called **ripple carry adder**.

# Finally getting close to a real computer

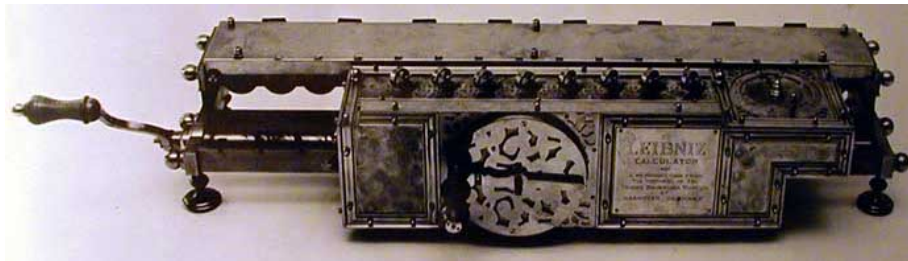
A Full Adder cell typically costs 16 transistors.





# From Pascal (1645) to Leibniz (1679)

Multiplication is just iterated additions:



# Cost of addition and multiplication

Think of what you do by hand: in terms of digit-level operations,

- addition is linear in the number of digits
- multiplication is quadratic in the number of digits.

The diagram illustrates the long multiplication of two 4-digit numbers,  $x_3x_2x_1x_0$  and  $y_3y_2y_1y_0$ . The multiplier  $y$  is written above the multiplicand  $x$ , with a multiplication symbol  $\times$  to the left. Four partial products are shown, each shifted one position to the left relative to the previous one. Each partial product is enclosed in a box and labeled with its corresponding digit indices:  $x_0y_3x_0y_2x_0y_1x_0y_0$ ,  $x_1y_3x_1y_2x_1y_1x_1y_0$ ,  $x_2y_3x_2y_2x_2y_1x_2y_0$ , and  $x_3y_3x_3y_2x_3y_1x_3y_0$ . These partial products are added together, indicated by plus signs. The final result is shown in a box at the bottom, labeled  $z_7z_6z_5z_4z_3z_2z_1z_0$ . The diagram demonstrates that the number of digit-level operations (represented by the boxes) grows quadratically with the number of digits.

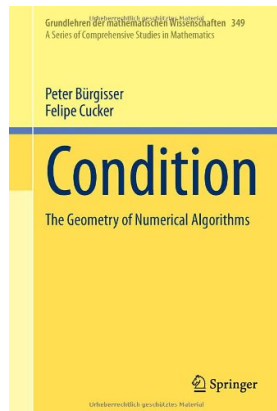
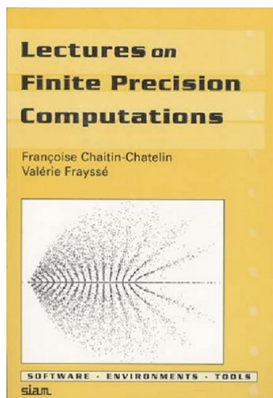
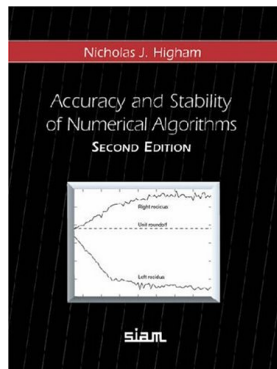
Multiplication can be reduced to subquadratic asymptotic cost (Karatsuba-Ofman, FFT), but it is practical for very large  $n$  only.

So a 100-digit multiplication is very expensive.

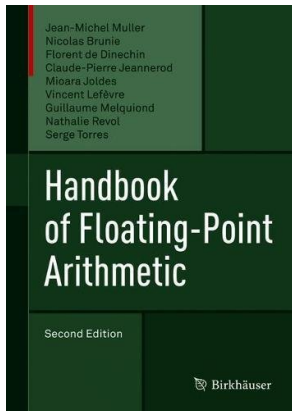
# Outline

- 1 Introduction
- 2 Number representations
- 3 Representing numbers to compute with them
- 4 Floating-point arithmetic**
- 5 Use of Computer Arithmetic: Historical Failures
- 6 Conclusion

# Bibliography



# Bibliography



What every computer scientist should know about floating-point arithmetic. David Goldberg. ACM Computing Surveys, 23(1):5–48, 1991.

# A famous failure: Patriot missile

- 1/10 is not representable by a finite number of digits in base 2

$$1/10 = 0.00011001100110011001100110011001100...$$

- On a 24 bit fixed-point register  
error =  $1.1001100... \times 2^{-24} \approx 0.000000095$  in decimal



- First Gulf War in 1991: an american Patriot missile battery failed to intercept an Iraqi Scud missile. The Scud killed 28 soldiers.
- After 100 hours, the error is about 0.34 s: a Scud travels at about 1500 m/s, it makes 500 m

# Can we count up to 6 with a computer?

$2 - 1$	1.0000000000000000
$\left( \frac{1}{\cos(100\pi + \pi/4)} \right)^2$	2.0000000000000111
$3 \frac{\tan(\arctan(10000))}{10000}$	2.999999999997162
$\left( \left( \left( \dots \left( \sqrt{\sqrt{\dots \sqrt{4}}} \right)^2 \dots \right)^2 \right)^2 \right)^2$ (20 times)	4.000000000629434
$5 \times \left\{ \frac{(1 + e^{-100}) - 1}{(1 + e^{-100}) - 1} \right\}$	NaN
$\frac{\log(e^{6000})}{1000}$	Inf

# Increase the precision?

Rump's polynomial :

$$f(x, y) = 333.75y^6 + x^2(11x^2y^2 - y^6 - 121y^4 - 2) + 5.5y^8 + \frac{x}{2y}$$

On a IBM 370, for  $x = 77617$  and  $y = 33096$ , computation of  $f(x, y)$  gives:

precision	computed value
single	1.172603
double	1.1726039400531
extended	1.172603940053178

But exact value is  $-0.827396059946821368141 \dots$



# Reproducibility

**Floating-point arithmetic:** operations suffer from rounding errors.

Floating-point operations (+, ×) are commutative but **not associative**:

$$(-1 + 1) + 2^{-53} \neq -1 + (1 + 2^{-53}) \quad \text{in double precision.}$$

**Consequence:** results of floating-point computations depend on the order of computation.

**Reproducibility:** ability to obtain bit-wise identical results from run-to-run on the same input data, with different resources.

# Motivations for reproducibility

Demands for reproducible floating-point computations:

- Debugging: look inside the code step-by-step, and might need to rerun multiple times on the same input data.
- Understanding the reliability of output.
- Contractual reasons (for security, ...).
- ...

# Sources of non-reproducibility

A performance-optimized floating-point library is prone to non-reproducibility for various reasons:

- Changing Data Layouts:

- Data
- Data partitioning,
- Data ordering,

- Changing Hardware Resources:

- Fused Multiply-Adder support,
- Intermediate precision (64 bits, 80 bits, 128 bits, etc),
- Data path (SSE, AVX, GPU warp, etc),
- Cache line size,
- Number of processors,
- Network topology,
- ...

# Compiler options

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

```
int main(){
```

```
volatile double a,b;
```

```
double x,y;
```

```
a=1.0/3;
```

```
b=0.1/2048;
```

```
x = a + b;
```

```
y = (a * (a + b)) + b;
```

```
if (y==0.0){
```

```
    printf("Error\n");
```

```
}
```

```
else {
```

```
    printf("Correct\n");
```

```
}
```

```
return 0;
```

```
}
```

```
> gcc -O3 -o repro repro.c
```

```
> ./repro
```

Correct

```
> gcc -O3 -ffast-math -o repro repro.c
```

```
> ./repro
```

Error

# Representation

A floating-point number  $x$  is represented in base  $B$  by:

- its **sign**  $s_x$  (0 for  $x$  positive, 1 for  $x$  negative)
- its **mantissa (significand)**  $m_x$  of  $n + 1$  digits
- its **exponent**  $e_x$ , an integer of  $k$  digits between  $e_{\min}$  and  $e_{\max}$

such that

$$x = (-1)^{s_x} \times m_x \times B^{e_x}$$

with  $m_x = x_0.x_1x_2 \dots x_n$  where  $x_i \in \{0, 1, \dots, B-1\}$ .

To obtain the best possible accuracy with a fixed-length mantissa  $n$ , the mantissa is **normalized**, that is to say,  $x_0 \neq 0$ ,  $m_x \in [1, B[$ .

A **special code** is necessary for 0.

# Before 1985

The result of an operation may differ depending on the language, the compiler and the computer architecture.

computer	B	n	$e_{\min}$	$e_{\max}$
Cray 1	2	48	-8192	8191
	2	96	-8192	8191
DEC VAX	2	53	-1023	1023
	2	56	-127	127
HP 28 et 48G	10	12	-499	499
IBM 3090	16	6	-64	63
	16	14	-64	63
	16	28	-64	63

Impossible to write **portable** numerical code.

# Revolution in 1985: IEEE-754 standard

A **standard** defines the representation of data and the behavior of basic operations in floating-point arithmetic

This standard defines:

- **formats** for data
- **special values**
- **rounding modes**
- **accuracy** of basic operations
- rules for **conversion**

**754** ANSI/IEEE Standard for Binary Floating-Point Arithmetic in 1985

**854** ANSI/IEEE Standard for Radix-Independent Floating-Point Arithmetic in 1987 (where  $B = 2$  or  $10$ )

**IEEE 754-2008**, ANSI/IEEE Standard for Binary Floating-Point Arithmetic in 2008

# Advantages of IEEE-754 standard

- make possible to write **portable** programs
- make programs **deterministic** from one computer to another
- keep **mathematical properties**
- perform **correctly rounded** operations
- perform **reliable** conversions
- facilitate the construction of **mathematical proof**
- facilitate the use of **exceptions**
- facilitate **comparisons** between numbers
- support **directed roundings** useful for **interval arithmetic**



# IEEE-754 standard: formats

$$B = 2$$

format	number of bits			
	total	sign	significand	exponent
single precision binary32	32	1	(1 implicit +) 23 (fraction)	8
double precision binary64	64	1	(1 implicit +) 52 (fraction)	11

The **significand** (normalized)  $m_x$  of  $x$  is represented by  $n + 1$  bits :

$$m_x = 1. \underbrace{x_1 x_2 x_3 \dots x_{n-1} x_n}_{f_x}$$

where  $x_i$  are 0 or 1.

The fractionnal part of  $m_x$  is called the **fraction** (of  $n$  bits):  $f_x$ .  
We have:  $m_x = 1 + f_x$  and  $1 \leq m_x < 2$ .

# IEEE-754 standard: exponent

The exponent  $e_x$  is a signed integer of  $k$  bits such that

$$e_{\min} \leq e_x \leq e_{\max}$$

Different representations are possible:

- two's complement,
- sign and magnitude,
- **biased**.

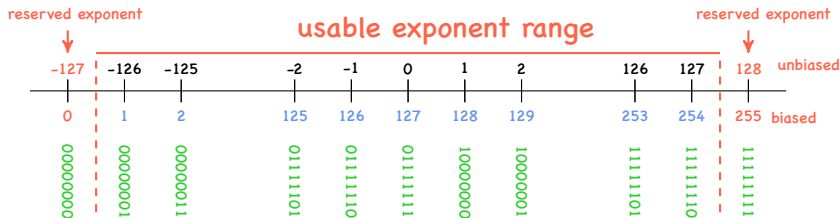
IEEE-754 standard: choice of a **biased representation** stored before the significand because it makes it possible comparisons between numbers in lexicographic order (forgetting the sign  $s_x$ ) and to represent 0 with  $e_x = f_x = 0$ .

The stored exponent is the **biased exponent  $eb$**  such that  $eb_x = e_x + b$  where  **$b$**  is the **bias**.

# IEEE-754 standard: reserved exponents

The unbiased exponent  $e_{\min} - 1$  and  $e_{\max} + 1$  (resp. 0 and  $2^k - 1$  with biased) are **reserved** for zero, subnormals special values.

format	size k	bias b	unbiased		biased	
			$e_{\min}$	$e_{\max}$	$eb_{\min}$	$eb_{\max}$
SP	8	127 ( $= 2^{8-1} - 1$ )	-126	127	1	254
DP	11	1023 ( $= 2^{11-1} - 1$ )	-1022	1023	1	2046



# IEEE-754 standard: representation of zero

$$e_0 = f_0 = 0, s_0 = ?$$

Two different representations:  $-0$  and  $+0$ .

Consistent with the fact that we have 2 distinct infinities. We have:  $\frac{1}{+0} = +\infty$  and  $\frac{1}{-0} = -\infty$ .

The standard enforces that the test  $-0 = +0$  returns True.

In single precision, the machine representation of  $+0$  and  $-0$  are:

x	$s_x$	$eb_x$	$m_x$
$-0$	1	00000000	000000000000000000000000
$+0$	0	00000000	000000000000000000000000

# IEEE-754 standard: special values

- The **infinities**,  $-\infty$  and  $+\infty$ :  $e_x = e_{\max} + 1$  and  $f_x = 0$ .
- **Not a Number**: NaN (exception, partial function)

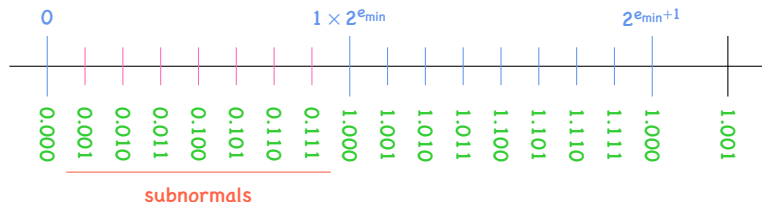
The result of an **invalid operation** such that  $0/0$ ,  $\sqrt{-1}$  or  $0 \times +\infty$ :  $e_x = e_{\max} + 1$  et  $f_x \neq 0$ .

NaN **propagates** during the computations.

In single precision, we have:

x	$s_x$	$eb_x$	$m_x$
$-\infty$	1	11111111	000000000000000000000000
$+\infty$	0	11111111	000000000000000000000000
NaN	0	11111111	000010010011110000000100 (for example)

# IEEE-754 standard: around zero, the subnormals



The smallest positive normalized number  $x$  is such that  $e_x = e_{\min}$ ,  $f_x = 0$ , and so  $x = 2^{e_{\min}}$ .

The "implicit 1" in the significand implies there is no floating-point number between  $0 \times 2^{e_{\min}}$  and  $1 \times 2^{e_{\min}}$  whereas there are  $2^n$  number between  $1 \times 2^{e_{\min}}$  and  $2 \times 2^{e_{\min}}$ .

The goal of **subnormal numbers** is to harmonize the repartition of floating-point numbers around 0.

We allow near 0 numbers with the form  $(-1)^{s_x} \times 0.f_x \times 2^{e_{\min}}$ :  
 $eb_x = 0$ ,  $m_x$  does not follow the rule of "implicit 1".

# IEEE-754 standard

## Subnormals and exponents

Exponent unbiased	Significand	Comment
$e_{\min} \leq e_x \leq e_{\max}$		normalized number, convention "implicit 1"
$e_x = e_{\min} - 1$	$m_x = 0$	zero
	$m_x \neq 0$	subnormal number, close to zero, no "implicit 1"
$e_x = e_{\max} + 1$	$m_x = 0$	infinity $(-1)^{s_x} \infty$
	$m_x \neq 0$	NaN, invalid operation, propagates during the computation, NaN $\neq$ NaN

The use of reserved exponent leads to costly computations.

# IEEE-754 standard: need for rounding

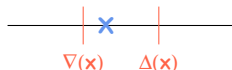
If  $x$  and  $y$  are 2 representable numbers, then the result of an operation  $\text{res} = x \odot y$  is not, in general, a representable number.

For example, in base  $B = 10$ , the number  $1/3$  is not representable with a finite number of digits.

It is necessary to **round** the result, that is to say to return one of the closest representable numbers.



# IEEE-754 standard: rounding modes



The norm proposes 4 rounding modes:

- rounding **toward  $+\infty$**  denoted  $\Delta(x)$  : return the smallest floating-point number greater or equal the exact result  $x$
- rounding **toward  $-\infty$**  denoted  $\nabla(x)$  : return the largest floating-point number less or equal the exact result  $x$
- rounding **toward 0**, denoted  $\mathcal{Z}(x)$  : return  $\Delta(x)$  for negative numbers and  $\nabla(x)$  for positive numbers
- rounding **to the nearest**, denoted  $\circ(x)$  : return the nearest floating-point number of the exact result  $x$  (breaks ties by rounding to the nearest even floating-point number)

The 3 first rounding modes are called **directed** rounding modes.

# IEEE-754 standard: correctly rounding

Let  $x$  and  $y$  be two representable number,  $\odot$  be on operations  $+$ ,  $-$ ,  $\times$ ,  $/$  and  $\diamond$  a rounding mode.

The standard requires that the result of the computation  $x \odot y$  be equal to  $\diamond(x \odot_{\text{exact}} y)$ . The result must be similar to the one obtain by computing with infinite precision and then rounding this result.

Similar for square root.

This property is called **correctly rounding**.

The standard describes an algorithm for addition, subtraction, multiplication, division and square root and requires that the implementation produces the same result as those algorithms.

# IEEE-754 standard: comparisons

The standard requires the comparison to be exact and not to overflow.

The specified comparison in the standard are:

- equality
- greater than
- less than

The sign of zero is not taken into account.

In case of a comparison with a NaN, the comparison returns False.

More precisely in case of equality: if  $x = \text{NaN}$  then  $x = x$  returns False and  $x \neq x$  returns True (equality is not reflexive but it is a way to detect a Nan)

# IEEE-754 standard: exception handling

No calculation should hinder the proper functioning of the machine. A mechanism with 5 flags makes it possible to inform the system about the behavior of operations:

**INVALID operation** the result by default is a NaN

**DIVIDE by ZERO** the result is  $\pm\infty$

**OVERFLOW** overflow toward  $\infty$ : the result is either  $\pm\infty$  or the greatest floating-point number (in absolute value) depending on the sign of the exact result and the rounding mode

**UNDERFLOW** overflow toward 0: the result is either  $\pm 0$  or a subnormal

**INEXACT** inexact result: raised when the exact result is not representable exactly. Returns the correctly rounded result by default.

# Summary of the IEEE-754 standard

	exponent	fraction	value
normalized	$e_{\min} \leq e \leq e_{\max}$	$f \geq 0$	$\pm(1.f)2^e$
denormalized	$e = e_{\min} - 1$	$f > 0$	$\pm(0.f)2^{e_{\min}}$
zero (signed)	$e = e_{\min} - 1$	$f = 0$	$\pm 0$
infinities	$e = e_{\max} + 1$	$f = 0$	$\pm \infty$
Not a Number	$e = e_{\max} + 1$	$f > 0$	NaN

format	# nb of bits	k	n	$e_{\min}$	$e_{\max}$	b
single precision	32	8	23	-126	127	127
double precision	64	11	52	-1022	1023	1023

value	single precision	double precision
greatest normalized > 0	$3.40282347 \times 10^{38}$	$1.7976931348623157 \times 10^{308}$
smallest normalized > 0	$1.17549435 \times 10^{-38}$	$2.2250738585072014 \times 10^{-308}$
greatest denormalized > 0	$1.17549421 \times 10^{-38}$	$2.2250738585072009 \times 10^{-308}$
smallest denormalised > 0	$1.40129846 \times 10^{-45}$	$4.9406564584124654 \times 10^{-324}$

# Accuracy of computations

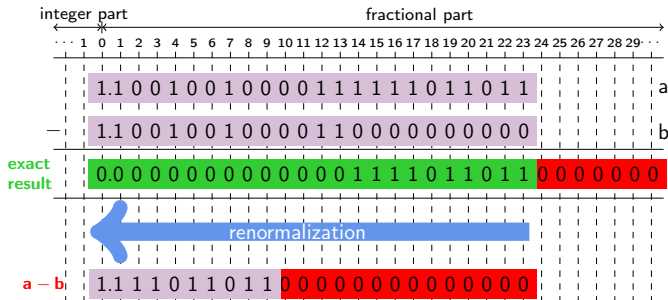
At each rounding, we lose a bit of accuracy, we call this **rounding errors**.

Even if an operation returns the best possible result (correctly rounded result), a sequence of computation can lead to huge errors due to the accumulation of rounding errors.

The two main sources of rounding errors during a computation are the **cancellation** and the **absorption**.

# Cancellation

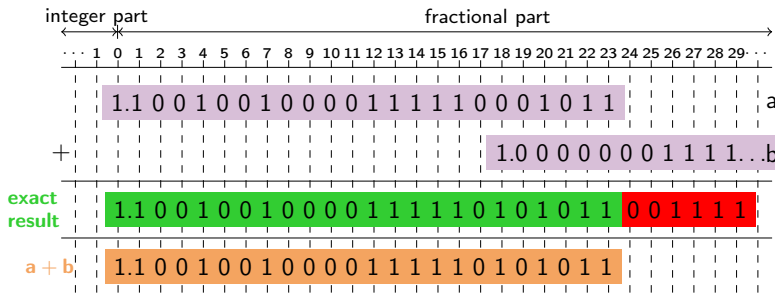
When we subtract two numbers very close.



If operands are results of previous computations with rounding errors, the 0's add on the right (red part) are wrong. The cancellation is said to be catastrophic when there is nearly no significant digits

# Absorption

When we add 2 numbers of different order of magnitude we can “lose” all the information about the smallest one.



Combine with a cancellation it can be catastrophic. for example in single precision, if  $a = 1$  and  $b = 2^{-30}$  then:  
 $a \oplus b = 1$  and so  $(a \oplus b) \ominus a = 0$ .



# Example of absorption

We compute, for large values of  $N$ , the sum:

$$\sum_{i=1}^N \frac{1}{i}$$

Result of a C program (in single precision) on a Pentium 4:

order	N			
	$10^5$	$10^6$	$10^7$	$10^8$
exact	1.209015e+01	1.439273e+01	1.669531e+01	1.899790e+01
$1 \rightarrow N$	1.209085e+01	1.435736e+01	1.540368e+01	1.540368e+01
$N \rightarrow 1$	1.209015e+01	1.439265e+01	1.668603e+01	1.880792e+01

# Floating-point numbers

Normalized floating-point numbers  $\mathbb{F} \subseteq \mathbb{R}$ :

$$x = \pm \underbrace{x_0.x_1 \dots x_{M-1}}_{\text{mantissa}} \times b^e, \quad 0 \leq x_i \leq b-1, \quad x_0 \neq 0$$

$b$  : basis,  $M$  : precision,  $e$  : exponent such that  $e_{\min} \leq e \leq e_{\max}$   
epsilon machine  $\varepsilon = b^{1-M}$

Approximation of  $\mathbb{R}$  by  $\mathbb{F}$  with rounding  $\text{fl} : \mathbb{R} \rightarrow \mathbb{F}$ .

Let  $x \in \mathbb{R}$  then

$$\text{fl}(x) = x(1 + \delta), \quad |\delta| \leq u$$

Unit rounding  $u = \varepsilon/2$  for rounding to the nearest

# Standard model of floating-point arithmetic

Let  $x, y \in \mathbb{F}$  and  $\circ \in \{+, -, \cdot, /\}$ .

The result  $x \circ y$  is not in general a floating-point number

$$\text{fl}(x \circ y) = (x \circ y)(1 + \delta), \quad |\delta| \leq u$$

IEEE 754 standard (1985 and 2008)

**Correctly rounded** : arithmetic ops  $(+, -, \times, /, \sqrt{\phantom{x}})$  performed as if first calculated to infinite precision, then rounded.

Type	Size	Mantissa	Exponent	Unit rounding	Interp
binary32	32 bits	23+1 bits	8 bits	$u = 2^{1-24} \approx 1,92 \times 10^{-7}$	$\approx 10^{\pm 126}$
binary64	64 bits	52+1 bits	11 bits	$u = 2^{1-53} \approx 2,22 \times 10^{-16}$	$\approx 10^{\pm 1023}$

Useful notation :  $\gamma_n := nu/(1 - nu) \approx nu$

# Advantages of the standard

IEEE arithmetic is closed: every operation produces a result.  
Default results:

Exception type	Default result
Invalid operation	NaN (Not a Number)
Overflow	$\pm\infty$
Divide by zero	$\pm\infty$
Underflow	subnormal numbers
Inexact	correctly rounded result


NaN is generated by operations such as  $0/0$ ,  $0 \times \infty$ ,  $\infty/\infty$ ,  $(+\infty) + (-\infty)$  and  $\sqrt{-1}$ .

Infinity symbol satisfies  $\infty + \infty = \infty$ ,  $(-1) \times \infty = -\infty$  and  $(\text{finite})/\infty = 0$ .

# Outline

- 1 Introduction
- 2 Number representations
- 3 Representing numbers to compute with them
- 4 Floating-point arithmetic
- 5 Use of Computer Arithmetic: Historical Failures**
- 6 Conclusion

# Therac 25: Description

- Therac 25:  
a radiation therapy machine
  - Production: 1980ies
  - Controlled by a DEC PDP-11
- 
- Two possible modes:
    - Electron beam: 5MeV beam on patient
      - Short time therapy
    - X-Ray beam: 25MeV beam on collider, X-Ray on patient
      - Long time therapy
  - One single electron gun, collider used for X-Ray mode.
  - Switching from one mode to another under software control

# Therac 25: A deadly failure

- Six high-overdoses between 1985 and 1987.
- ⇒ Therac 25 called back in 1987
- Particular cases:
  - Marietta, Georgia, June 1985: 15000rad instead of 200rad
    - Breast ablation necessary
    - Patient alive
  - Hamilton, Ontario, July 1985: 15000rad instead of 200rad
    - Patient dies
  - 4 other similar cases, even after hardware update
- What had happened?

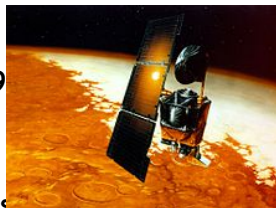
# Therac 25: Failure Analysis

- The failure could happen only in very rare circumstances
- Testing had not caught these cases
- When hardware detected wrong configuration of the system, software increased an error variable  $e$  by 1.
- The operator could start the radiation only if  $e$  was 0.
- The error variable  $e$  was stored on a 1 byte memory location
- After the 255th error, the next error made  $e$  become 0!
- Patients died due to a classical integer overflow problem



# Mars Climate Orbiter: Description

- **Mars Climate Orbiter:**  
a space mission sent to Mars in 1999
- Two modules:
  - Mars Climate Orbiter stays in orbit
  - Mars Polar Lander descends to mars



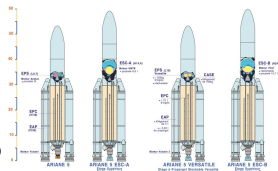
- **Mars Climate Orbiter was built by two companies**
- Lockheed delivered a gravitational-force measuring equipment
- Data was analyzed by a computer built by Jet Propulsion Lab

# Mars Climate Orbiter: Failure Analysis

- On 09/23/1999, Mars Climate Orbiter crashed on mars
- What had happened?
- The Lockheed system delivered data in fixed-point format
  - with imperial pound-force as a basic unit
  - $f = \beta_1^q \cdot \sum f_i \beta^i$
- The Jet Propulsion Lab computer expected fixed-point data
  - with metric Newton as a basic unit
  - $f = \beta_2^q \cdot \sum f_i \beta^i$
- Half a billion US \$s lost due to wrong fixed-point arithmetic

# Ariane 5: Description

- ESA builds Ariane space launchers
- Ariane 5 followed Ariane 4
- First Ariane 5 flight: June 4th 1996



- Development time for Ariane 5: 10 years
- Cost of Ariane 5 development: 7 billion US \$
- Ariane 5 is not a complete redevelopment  
Ariane 5 somehow just extends Ariane 4

# Ariane 5: half a billion US\$s lost

- At its first flight, Ariane 5 explodes
- The explosion happens 37 seconds after the launch



- The cost for this Ariane 5 and the satellites is roughly 500 million US \$
- Any problem at an inaugural launch yields to reputation problems for ESA

# Ariane 5: Failure Analysis

- Ariane 5 had a software module that check horizontal speed
- The software module was needed for the first seconds of flight
- Somewhere in that code (in ADA), there is this line:

```
horiz_veloc_bias := integer(horiz_veloc_sensor);
```

- This means **a floating-point value is converted to integer**
  - The conversion is not protected and may overflow
  - Testing had been done on Ariane 4
  - Ariane 4 is much slower than Ariane 5
- ⇒ **An overflow happened on Ariane 5 and cost 500 million US \$.**

# The Pentium Bug: Description

- Intel launches Pentium CPUs in 1993
- The old 486 floating-point unit gets reworked



- The 486 floating-point division used a restoring division
  - That division produced 1 bit per division cycle
  - The Pentium uses SRT-4, producing 2 bits per division cycle
  - There are less operations between each division cycle
- ⇒ The Pentium `fdiv` is about 5 times faster

# The Pentium Bug: a billion US\$ lost

- Pentium is launched in 1993
  - In June 1994, a researcher discovers a bug in `fddiv`
    - He was doing research on primality
    - Some basic facts about integer division were not satisfied
  - Intel starts by denying the problem
  - Eventually, Intel admits that `fddiv` is bugged
    - About 4 leading digits are correct
    - Then, the divider might produce nonsense
  - With market pressure, Intel exchanges all buggy CPUs for free
- ⇒ about a billion US \$s of loss for Intel

# The Pentium Bug: Analysis

- What was wrong with the Pentium `fdiv`?
- The SRT-4 algorithm uses a table
  - One digit of the residual and one digit of the divisor
  - give a guess on one digit of the quotient
- Generation of the right table is pretty hard
  - but well understood in theory
- What happened:
  - The design team computed a table with 1066 entries
  - Only 1061 entries were transferred to the HW architects
  - A for loop was 5 iterations short
- Morality: prove your implementation not just the algorithm



# Patriot Missiles: US soldiers killed

- MIM-104 Patriot missiles used by NATO
  - Patriot supposed to intercept enemy missiles
  - Used e.g. in the 1st Gulf war in 1991
- 
- Interception capability based on high performance radar system
  - Missile able to find/follow target if target is close enough
  - i.e. launch unit must fire the missile with enough precision
  - On 02/25/1991, a Patriot missile fails to intercept an Iraqi SCUD missile
  - The SCUD missile hits a US garrison in Saudi-Arabia, killing 28

# Patriot Missiles: Analysis

- What had happened?
  - The Patriot HW clock delivers time in 1/10ths of seconds
  - The guidance software handles time in seconds, using single precision floating-point
  - Conversion from HW time is not done dividing by 10 but by multiplying by 0.1
  - 0.1 is not representable in binary floating-point
  - A relative error of about  $5.96 \cdot 10^{-8}$  occurs
  - The Patriot system had been running for more than 100 hours
- ⇒ Time was off by:  $10 \cdot 100 \cdot 3600 \cdot 5.96 \cdot 10^{-8} \text{ s} = 0.21 \text{ s}$
- In these 0.21 s, a SCUD missile flies about 360 meters
  - The Patriot missile cannot follow missiles that far.

# Vancouver Stock exchange: Description

- Vancouver, BC, Canada, has a stock exchange since 1906
- In 1982, the Vancouver Stock Exchange adopts an index
- A stock exchange index is a weighted sum of traded stocks:

$$I = \sum_i w_i \cdot c_i$$

- To avoid recomputation of the whole index each time a stock changes, **perform an update of the index**:

$$I' = \sum_i w_i \cdot (c_i + \Delta_i) = I + \sum_i w_i \cdot \Delta_i$$



# Vancouver Stock exchange: Where's the money?

- The Vancouver Stock Exchange Index starts in January 1982
- The weights are set such that the index starts at 1000 points
- Economy throughout 1982 and 1983 was good, trading was at high level
- In November 1983, the Vancouver Stock Exchange Index was at 524 points
- External advisors from Toronto and California reevaluated the index to 1099 points.

# Vancouver Stock exchange: Failure Analysis

- What had happened?
  - At each update, the product  $w_i \cdot \Delta_i$  and the new index sum  $I + w_i \cdot \Delta_i$  was computed on 4 valid digits
  - Instead of unbiased rounding, **truncation** was used
- ⇒ each update yielded a **systematic negative error**

$$\tilde{I}' = \nabla(I + w_i \cdot \Delta_i) = I' + \delta$$

where  $10^{-4} < \delta \leq 0$ .

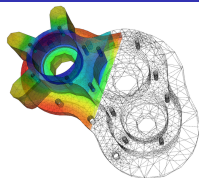
- At 2900 transaction a day in 1982, and 440 days of wrong computations, we get an overall mean error of:

$$5 \cdot 10^{-4} \cdot 2900 \cdot 440 = 638$$

- **Morality: the error of long running numerical systems must be shown to tend to zero.**

# Finite Elements: A method

- The **Finite Element Method** is a numerical technique for **solving ODEs**.
- Example: flow of air in atmosphere
- Idea behind the Finite Element Method:
  - Decompose the domain into small, but finite elements (boxes)
  - Suppose the solution (function) to be constant in each element
  - The ODE problem, with its boundary and starting values, becomes a linear system
  - Solve the linear system and hence obtain an approximate solution to the ODE.
- **Using finite elements instead of infinitely small ones yields error.**
- **The analysis and bounding of such method error is important.**



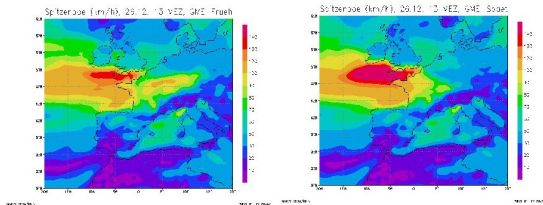
# Finite Elements failure 1: Tacoma Bridge

- Washington, US West Coast:  
The Tacoma Narrows Bridge connects Tacoma with the Kitsap Peninsula
- First Tacoma Bridge built in July 1940
- The bridge collapses in November 1940.
- What had happened?
- Finite Element Method at the time had modelled all static forces
- The Finite Element Method was proven correct if all dynamic forces vanished.
- The slightest wind creates a dynamic force onto the bridge structure.
- Morality: always check your hypotheses with reality.



# Finite Elements failure 2: Lothar storm

- Meteorological services use FEM a lot
- On 12/24/1999 1:00PM, German Meteorological Service predicts a storm
- Then, to be sure, adds data and recomputes
- On 12/24/1999 4:30PM, the service cancels the alert
- The storm hits Germany badly on 12/25/1999.
- Some data for the Atlantic was just interpolated.
- Morality: when it comes to safety-critical problems, classical numerical methods and Computer Arithmetic are not enough.





# Is testing a solution?

- Smart Software testing may enhance safety of the system
- However, we need to be sure that we test with real-life values
- **Testing** for numerical systems is **not** (always) **worthwhile**
  - A double precision floating-point variable can represent about  $10^{19}$  different values
  - Current computers manipulate about  $10^{15}$  such variables **per second**
  - A failure might occur for **1** out of the  $10^{50}$  possibilities
- **Multiple sources** of approximation **error**:
  - A measurement approximates a physical value
  - The mathematical model approximates reality
  - The code approximates functions with polynomials
  - Floating-point arithmetic approximates the reals ...
  - **Humans can only think of about 8 things at a time**

# We need proofs & code generation

- Numerical code can be made safe
  - if modelization and software development care about errors in an a priori way
  - i.e. if every values comes with its error analysis and its bounds.
- Proofs of correctness are needed
  - They are easier to establish when made while developing the software
  - Humans may forget to check corner-cases in proofs
  - Computer checked proofs are becoming accessible
- Humans can think of 8 things at a time
  - Do no longer write numerical code by hand
  - Have the computer generate numerical code for you
  - A research project for the next decade

# Outline

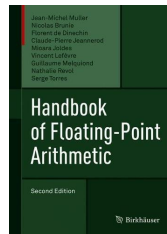
- 1 Introduction
- 2 Number representations
- 3 Representing numbers to compute with them
- 4 Floating-point arithmetic
- 5 Use of Computer Arithmetic: Historical Failures
- 6 Conclusion**

# Conclusion

- Number representation systems
  - Different radices  $\beta$  possible and used
  - Use a number representation that fits your technology!
  - Redundant number systems enable dirty but fast algorithms
- Operations
  - Addition is typically  $\mathcal{O}(n)$
  - Multiplication is naively  $\mathcal{O}(n^2)$
- Fixed & Floating-Point numbers are not the same as the reals
  - In Fortran, we write  
`real*8 :: a, b, c`
  - But:  $a \oplus (b \oplus c) \neq (a \oplus b) \oplus c$
- Correct modeling, tidy error & round-off analysis are required
  - You don't want to be the engineer responsible of a failure
- Computer checked proofs are helpful and accessible

# Read the good books

**Handbook of Floating-Point Arithmetic,**  
by Muller et al.



Also

- Digital Arithmetic by Ercegovic and Lang
- Arithmétique des ordinateurs by J.-M. Muller, (on his page)
- What Every Computer Scientist Should Know About Floating-Point Arithmetic by Goldberg  
(Google will find you several copies)
- The web page of William Kahan at Berkeley.