Floating point calculations

Daniele Petrili
Jana Fank
Christoph Gehrlein

AGENDA

1	**Definition:** Floating Point Numbers
2	**Representation:** Floating Point Numbers
3	**Problems:** Floating Point Numbers
4	**Solution:** Kahan Summation
5	**Optimization:** Sorted floats vs. unsorted floats

AGENDA

1	**Definition:** Floating Point Numbers
2	**Representation:** Floating Point Numbers
3	**Problems:** Floating Point Numbers
4	**Solution:** Kahan Summation
5	**Optimization:** Sorted floats vs. unsorted floats

Definition: Floating Point Numbers

Floating-point numbers =
common way to represent decimal numbers - **real numbers** for the computer.

Real numbers =
numbers that have decimal places => -7.25

completely unproblematic: 7/2 = **3.5**
more difficult: 10/3 = **3.33333...**

=> A computer can not easily view an infinite number, or calculate.

Definition: Floating Point Numbers

Each number consists of a sign, a part before the comma and the numbers after the decimal point:

Example: -5432578865,7890763257556

=> Sign: -
=> Before Comma: 5432578865
=> Decimal: 7890763257556

Exponential notation: -5,432*10^9 => Sign*Mantissa*Base^Exponent

AGENDA

1	**Definition:** Floating Point Numbers
2	**Representation:** Floating Point Numbers
3	**Problems:** Floating Point Numbers
4	**Solution:** Kahan Summation
5	**Optimization:** Sorted floats vs. unsorted floats

Representation: Floating Point Numbers

IEEE-Standard =
defined standard for binary floating-point representations in computers and lay down detailed procedures for performing mathematical operations, in particular for rounding, determine.

x = s * m * β ^ e => Sign s (**1** Bit)
Mantissa m (**p** Bits)
Base β (assumed to be even)
Exponent e (**r** Bits)
e_{max},e_{min}(allowable exponents)

Representation: Floating Point Numbers

x = s * m * β ^ e =>

Sign s (**1** Bit)
Mantissa m (**p** Bits)
Base β (assumed to be even, normalized β=2)
Exponent e (**r** Bits)
e_{max}, e_{min}(allowable exponents)

IEEE Types for example single and double:

typ	size (1+r+p)	exponent (r)	mantissa (p)	values of the exponent (e)	bias value (B)
single	32 bit	8 bit	23 bit	-126 ≤ *e* ≤ 127	127
double	64 bit	11 bit	52 bit	-1022 ≤ *e* ≤ 1023	1023

Representation: Floating Point Numbers

Decimal Number => IEEE-Standard (Example for single typ)

- 5.25

1) Sign => negativ = **1**
2) Number in binary => 5.25 = 101.01
3) Move decimal point so that the first number is a one => 1.0101
4) Record number of displaced points (n)
5) Calculate exponent: Bias + n = 127 + 2 => 129 = **10000001**
6) Calculate mantissa shifted partial (minus 1) => 1.0101 -1 = **0101**

AGENDA

1	**Definition:** Floating Point Numbers
2	**Representation:** Floating Point Numbers
3	**Problems:** Floating Point Numbers
4	**Solution:** Kahan Summation
5	**Optimization:** Sorted floats vs. unsorted floats

Problems: Floating Point Numbers

Floating-point numbers are only approximations!

In most cases approximations is not a problem.
The problem, however, it is if they create a sequence errors.

10/3 = 3.3333 => 3.3333 x 3 = 9.9999

9.9999 is not 10!

Problems: Floating Point Numbers

Relative Error and Ulps

Example: with β =10 and p=3

3.12 x 10^{-2} =0.0314 => Error by 2 units in the last plasce =ulps

Representing 0314159

3.14 × 10-2, => Error by 159units in the last place=ulps

- Relative error $((\beta/2)\beta^{-p})$
  - Simplify the difference between two numbers divided by the real number

Example: 3.14159 by 3.14 × 10^{0}

Relative error: .$(10/2)10^{-3}$)= .005

Problems: Floating Point Numbers

Guard Digits

- Most computer systems

Given $0.100 \times 2^1 - 0.111 \times 2^0$
$0.1000 \times 2^1 - 0.0111 \times 2^0$ ===> 0.0001×2^1

Without guard digit:
0.001×2^1 ---Relative error of 1

- Relative Error can be large without

Problems: Floating Point Numbers

More problems

- Cancellation (Catastrophic, Benign)
  Operands are subject to rounding errors

- Exactly Rounded Operations
  {0, 1, 2, 3, 4} round down, {5, 6, 7, 8, 9} round up

  Numbers ending in 5:
  rounded result have its least significant digit be even

AGENDA

1	**Definition:** Floating Point Numbers
2	**Representation:** Floating Point Numbers
3	**Problems:** Floating Point Numbers
4	**Solution:** Kahan Summation
5	**Optimization: Sorted floats vs. unsorted floats**

Solution: Kahan Summation
What is it?

- **What**? Computes the sum of n floating point numbers
- **Why**? Rounding error occurs through finite precision floating point numbers (limited number of digits, example on the next slides)
- **How**? Reduces the numerical error by a compensated summation:
  The algorithm uses feedback from the previous iteration and evaluates an error which is added to the sum in the following step

Solution: Kahan Summation

The algorithm

```
v = [a;b;c;...;n];    % define a vector of n numbers you want to add

sum_kahan = 0;  % initialize the Kahan sum
e = 0;                % initialize the error
n = size(v,1);        % returns the number of rows

for i = 1:n                                % for loop computes a + b + c + ... + n
      y = v(i,1) - e;                      % consider the error of the previous step
      temp = sum_kahan + y;                % compute the new sum
      e = (temp - sum_kahan) - y; % compute the new error (lost digits)
      sum_kahan = temp;                    % assign the sum to final variable
end
```

Solution: Kahan Summation
Matlab Example

- Compute the following **sum** of 3 floating numbers:
  - 10.000,0 + 3,14159 + 2,71828

- Suppose we are using **six-digit** decimal floating point arithmetic
  → e.g. 10.000,08 has 7 digits and will be rounded to 10.000,1

- **Problem**: Adding 2 small numbers to a much larger number leads to a loss of many low order digits

- **Solution**: Improve the accuracy by using Kahan Summation
  → continuously recovering the lost bits from the previous addition and add this “error” to the sum in the next iteration

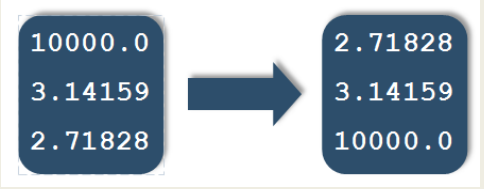
AGENDA

1	**Definition:** Floating Point Numbers
2	**Representation:** Floating Point Numbers
3	**Problems:** Floating Point Numbers
4	**Solution:** Kahan Summation
5	**Optimization: Sorted floats vs. unsorted floats**

Optimization: Sorted vs. unsorted floats

Basic ideas and Matlab Code

- **Idea**: Sort the floating numbers from small to large
- **Why**? We don´t loose digits and thereby accuracy if we add the smaller numbers first and round them afterwards.
- In the previous **example** it would look like that:



- The result in Matlab shows that now the calculation without Kahan's algorithm has the same accuracy as the one using the algorithm (for the choice of these numbers)

Thank you very much
for your attention!

Any Questions?