Computational Methods

H.J. Bulten, Spring 2017

www.nikhef.nl/~henkjan

Lecture 1

H.J. Bulten henkjan@nikhef.nl

website: www.nikhef.nl/~henkjan , click on Computational Methods

- Aim course: practical understanding of numerical methods in (physics) problems.
  - Numerical calculations: large scale, not analytically solvable problems
  - techniques: integration, Fourier analysis etc.
  - are the results correct? Verify. Always try to validate your code.
- Contents course:
  - Hands-on techniques in practice: book: *Numerical Recipes,*

    Press, Teukolsky, Vetterling and Flannery
  - Version 1 available on web: www.nr.com
  - covered material: see next slide
  - techniques to perform numerical tasks, C++ (version 3)
  - extra material : FORM?, fitting?, Schroedinger equation, Monte Carlo
- Examination: Completion of exercises. Mail the source code and a small (1 page) summary of your findings.

Course contents

- Introduction: numerical errors, machine representations, (programming), CIC filter

Interpolation and extrapolation,	chapter 3
Integration of functions,	chapter4
Evaluation of functions,	chapter 5
Roots, Minima and maxima,	chapters 9 and 10
Fourier methods, wavelet transforms	chapters 12 and 13
Systems of linear equations,	chapter 2
Eigensystems,	chapter 11
Schroedinger equation	
Integration of differential equations,	chapter 17
Boundary value problems,	chapter 18
Monte Carlo techniques	(chapter 7)

Examination

- Exercises
  - At least 1 exercise per week. The larger exercises will get a higher weight in the final grade. Code and output must be mailed to henkjan@nikhef.nl . I will discuss the exercises during the course (except for the final one).
  - Please, include your student number + institute in the first mail.
  - Exercise 1 of today has weight factor 1 (lowest). It is meant to verify that you can compile a simple piece of code, and to familiarize you with the problem of stability of numerical algorithms.
  - Exercise 2 has a weight factor of 2 points. It demonstrates the use of integer arithmetic to cancel numerical errors.
  - In the discussion of the exercises, I will use the routines from edition 3 (nr3.h etc) which come with the book and can be found on the web.
  - A full copy of the numerical recipes include files is available from www.nikhef.nl/~henkjan/CODE/allincludes.tar
  - It is a file created with tar, to extract the files use :

    tar -xf allincludes.tar

Computational Methods

Field of Computational Methods

- non-analytical problems (else solve analytically)
- computing uncertainties – controlling accuracy

optimize algorithms

- speed
- accuracy
- stability
- applicability
- maintainability
- elegance
- simplicity
- ……..

Verification

- Larger codes : **verify**
  - Calculate a trivial example (known analytically)
  - Vary input parameters
    - e.g. Monte Carlo: try several runs with 100000 events
      - Do the results match statistically?
    - e.g. differential equations: change input parameters slightly
      - Do the results change linearly? Chaotically?
  - Compare a single case to e.g. Wolfram:
    - e.g. your code contains a billion integrals. Calculate one explicitly and compare to the calculation in Wolfram
    - www.wolframalpha.com
  - In case of discrepancies: use **debugger**.

Uncertainty, integers

Computational error sources:

- integers: fixed point
  - int, long, short, char
  - signed/unsigned
  - accurate:
    - division – skip remainder
    - underflows/overflows
    - Banking
- Widely used: two's complement.
  - e.g. for two-byte signed integers (shorts) - run from -32768 to +32767
    - 0 = 0x0 = 0000000000000000
    - 2^16-1 = 0x7fff = 0111111111111111
    - -2^16 = 0x80 = 1000000000000000
    - -1 = 0xffff = 1111111111111111
    - Except for overflows (fixed multiples of 2^16) the differences are correct both for unsigned and signed integers: -1 + 1 = 0xff + 0x01 0x(1)00 = 0), addition and multiplication can be implemented with the same coprocessor steps for positive, negative, and unsigned integer arithmetic.

uncertainty

Computational error sources:

- floating point
  - float, double, real*4,real*8, complex*16
  - machine-dependent representation
  - signbit,mantissa,exponent
  - base usually 2, sometimes 16

$$x = s\,m\,B^{e-E};\quad s = positive(s=0)\ \vee\ negative(s=1)$$
$$m = mantissa(1010000 = 1/2 + 0/4 + 1/8 + 0/16 \ldots 0/2^{N})$$
$$B = base\ \ usually\ \ 2\ \ sometimes\ \ 16$$
$$E = \ offset\ \ exponent$$

- e.g. ½ = 0 1000000.. 1000..
- 3=0 1100000.. 10..010
- leading 1 may be omitted in some representations
- addition: left-shift mantissa, until exponents equal
  - Loss of accuracy. Adding numbers with different exponents leads to a loss of significant number of bits.

floating point representation

smallest non-negative number – depends on exponent (and definition of mantissa)

largest representable number: depends on exponent

- Overflows: e.g 500!, exp(5000), and underflows (exp(-5000)), must be captured! (in your code. Do not rely on compiler)
  - Mathematical functions from libraries expect input that can be represented. Especially for complex arithmetic errors are easily made:
    - e.g. in calculating the norm of a complex number, the squares of the real and imaginary parts may be added. If this is done incorrectly {Norm(a+ib) = sqrt(a*a+b*b)} then the maximal allowable value for a and b are ½ sqrt(max_float) instead of ½ (maxfloat). In your code, you should always capture possible overflows.

$$c = a + ib$$

$$|c| = \sqrt{a^2 + b^2}$$

$$\sqrt{a*a+b*b} \rightarrow \left(a^2+b^2 < Maxfloat \;\; ; \;\; a,b < \sqrt{Maxfloat/2}\right)$$

$$a\sqrt{1+b/a \;\; b/a} \;=\; b\sqrt{1+a/b \;\; a/b} \rightarrow a,b < Maxfloat/2$$

floating point errors

- round off errors:
  - machine + compiler dependent
  - typically precision 10^{-16} double, 10^{-7} float
- round-off always contributes to error
  - Lucky ?

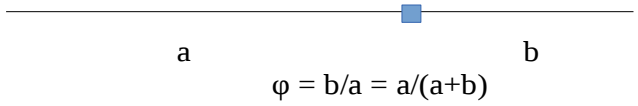
$$\epsilon \sim \epsilon_{machine} \sqrt{(N)}$$

- Multiplication: add exponents, multiply mantissae : keeps machine accuracy (the product of two floating point numbers has the same number of significant bits as a single floating point number)
- Summation: makes mantissa equal. If the mantissa differs by n bits, the precision of one of the two numbers is reduced by n bits.
  - Extreme case: $1 + 10^{-18} = 1$ (for typical 8-byte double precision representation).

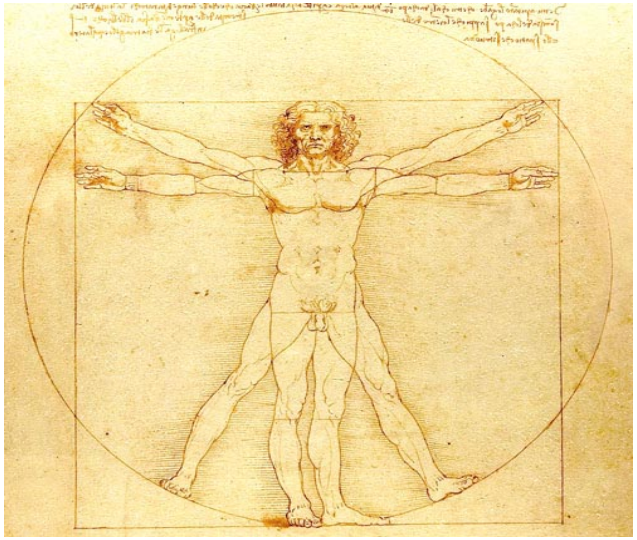
Stability

- Stability: round off error may accumulate:
- e.g. golden section:

$$\phi = \frac{\sqrt{5}-1}{2}, \quad \phi^{n+1} = \phi^{n-1} - \phi^{n}$$



φ = b/a = a/(a+b)



- example : phi has roundoff error. The zeroth term is exact (1) but the first term is a little bit too small or too large
  - error blows up exponentially (verify!)
  - Exercise to see how this works.
  - Exponentially increasing errors: Typical behavior encountered in e.g. Bessel functions. Check your code for these types of numerical errors.

Exercise – stability of the golden mean

Exercise 1. Determination of stability of golden mean approximation.

Powers of the golden mean, $\phi = \frac{\sqrt{5}-1}{2}$,

can be calculated using the following recursion relation:

$$\phi^{n+2} = \phi^{n} - \phi^{n+1}.$$

However, this relation is numerically unstable.

Determine, by comparing the result of this recursive relation for higher powers of phi to the direct calculation (obtained by $\phi^n = e^{n \ln \phi}$), after how many terms the difference between those two calculations is larger than 0.1% and after how many terms it is larger than 50%. Give the values of n and ϕ^n, calculated with the recursion relation and with the direct method, both for single precision (4 bytes) and double precision (8 bytes).

The direct calculation (using *pow(phi, n)* or *exp(n log(phi))*) gives an error close to the machine accuracy, which may be verified by dividing the result of the direct calculation n times by the calculation of ϕ. The latter operation (n successive divisions or multiplications) will result in a fractional error that is approximately equal to n times the machine accuracy, since the relative error grows linearly in each step (Why?). This accumulated error, although larger than the error in n random multiplications, is still much smaller than the error obtained by the recursion relation.

Do you understand the results?

This exercise has the lowest weight of 1 point. Mail the source code and your answer to henkjan@nikhef.nl before Thursday Feb 9, 0:00; the results will be discussed next lecture.

Conversions, data types

- Sometimes it is necessary to manipulate bits, or to change from floating point to integer and vice versa.
- Example:

```
#include <stdint.h>
#include <iostream>
#include <iomanip>
using namespace std;

int main() {
   uint32_t k=0x1020304; // assign an unsigned, 32 bits integer to hexagonal 0x1020304
   double l;
   l = (double) k; // convert the 32-bit integer to a 64 double with the same numerical value. The
converse is also possible  (if l is small enough) : k = (uint32_t) l;
   uint64_t *m,n; // m points to 8 bytes, interpreted as 64-bits unsigned integer
   m = static_cast<uint64_t *>((void *)&l); // static cast from double* to integer* is not allowed.
   n= (0xff); //least significant byte =1111111, n=255;
   for (int i=0; i<8; i++) {
         cout <<i<<"-th significant byte " << std::hex << (*m&n) << endl;
         *m=(*m>>8); // right-shift 8 bits
   }
   return 1;
}
```

CIC filter

- Integer arithmetic: exact
  - Nice for Cascaded Integrator Comb filters – CIC filters.
  - The problem: digital sampling: each sample has noise.
  - e.g. (expensive) fast ADC might sample with 1 MHz and have 20 bits resolution, lowest 2 bits noise. That means a resolution of about 4 ppm full scale per sample.
  - Data needed at lower frequency, e.g 1 kHz. Then oversampling helps: for white noise, each factor of 4 oversampling gives a factor of 2 better resolution. Oversampling with 1024 times will gain a factor of 32 better resolution, or 60 dB!
  - Fast sampling: lots of data and fast computing needed to combine the data.
  - Real-time front-end downsampling: fast and cheap algorithms needed.
  - CIC filter: moving average over last M samples. If M is a power of 2, then you can just add the last M samples and divide by right-shifting $^{2}\log(M)$ bits. So, a moving average can be done by e.g. adding 1024 samples and then right-shifting the result 10 bits.
  - It becomes more efficient if, for each sample i, you add the current sample i to the sum and subtract sample i-M. Then, to calculate the averages over each sample, you only have to do 1 addition and 1 subtraction (plus a right-shift to keep the normalization correct) instead of M additions. However, you have to keep the last M samples in memory, else you cannot subtract sample i-M

Homework Exercise: CIC filter

- CIC filter:
  - Moving average (digital signal processing techniques)
  - https://en.wikipedia.org/wiki/Cascaded_integrator–comb_filter

$$y[M-1] = \sum_{0}^{M-1} x_i$$

$$y[2M-1] = \sum_{M}^{2M-1} x_i$$

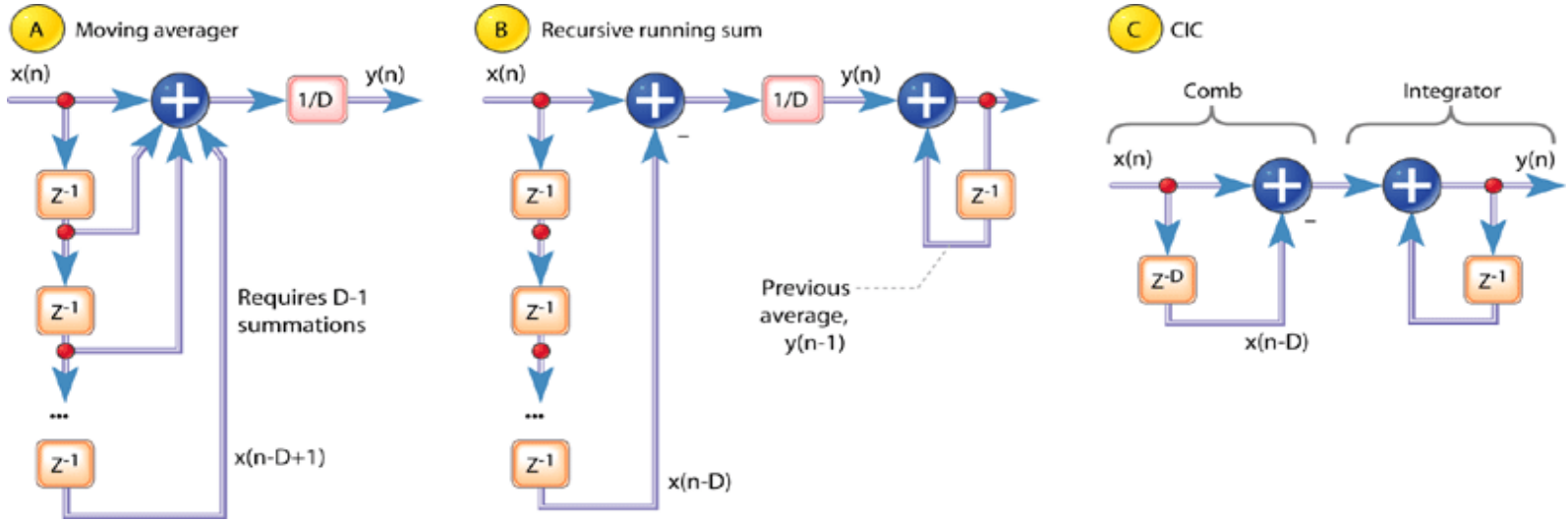
Moving averages after M and 2M samples (before dividing by M)

$$y[2M-1] = \sum_{0}^{2M-1} x_i - y[M-1]$$

The moving average after NM samples can be calculated by subtracting from the current the result of (N-1)M samples

  - Response moving average: the average value over the last M samples is taken. This means, that noise with frequency components with frequencies (M, M/2, .. 1)/dt are exactly cancelled, and arbitrary high frequencies >1/dt are averaged and downsampled. e.g. a frequency that fits 17.3 times in M samples contributes only about 0.3/17.3 to the downsampled signal.
  - CIC filters can be cascaded, you can do e.g. 4 in a row. The first one sums the input, the second works on the sum of the first, etc. This yields better high-frequency noise subtraction than a single moving average filter.

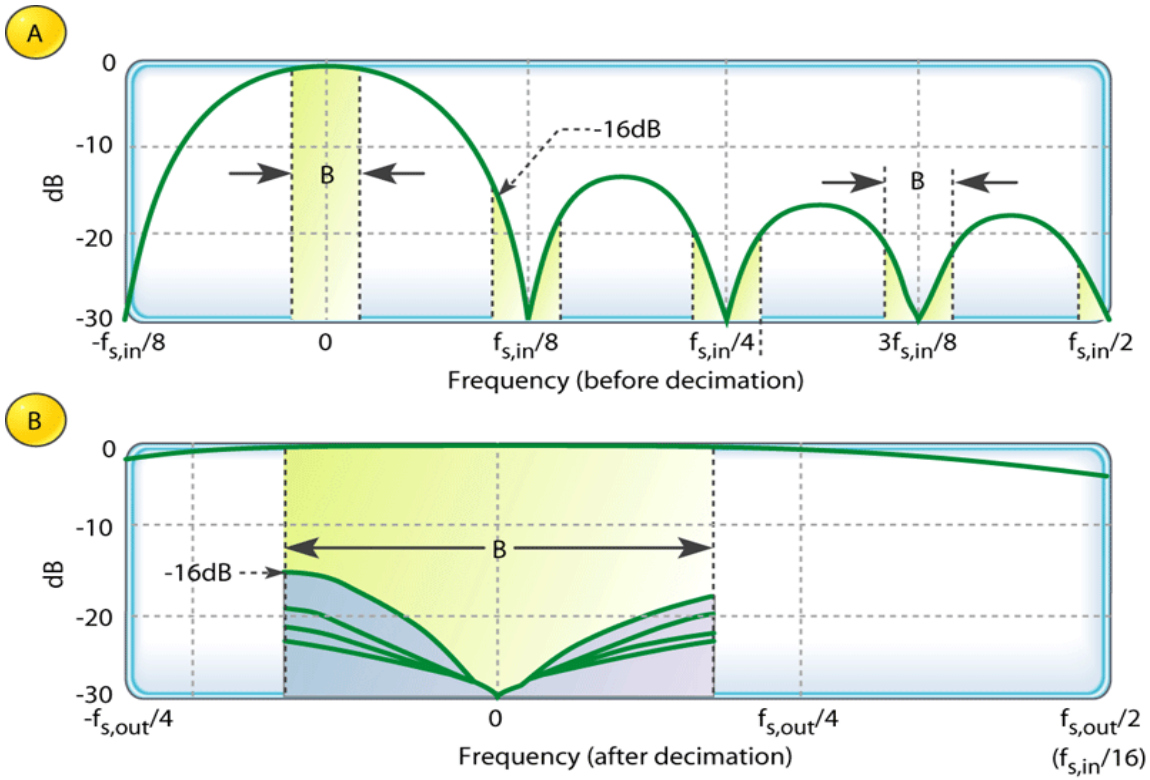
CIC filter



- CIC filter:
  - Integrator: addition of the new sample to the sum
  - Comb: subtraction of sample i-M from the sum
  - Next performance gain: you can do the subtraction (the Comb part) AFTER decimation. This means : you just keep adding samples at high rate to the sum. Every M-th sample, you subtract the previous sum obtained i-M samples ago. Normalization (dividing by M) is done at the end.

CIC filter

- Response CIC filter: see e.g. http://www.embedded.com/design/configurable-systems/4006446/Understanding-cascaded-integrator-comb
-



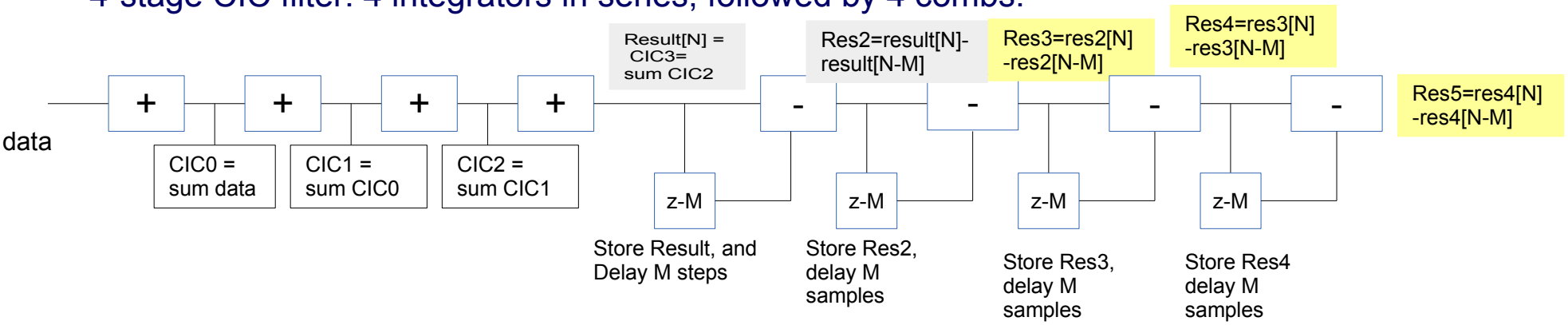
CIC filter, decimation with a factor 8. At the original sampling speed, the Nyquist frequency is $fs_{in}/2$. $fs_{out} = fs_{in}/8$ shows the response and the aliased, down-sampled strength from higher frequencies.

CIC filter

- A CIC filter can be efficiently implemented using integers. In this filter, the sums are stored and only the decimated sums need to be subtracted. This will work correctly as long as overflows of the integers are taken into account correctly.
- The integers must be large enough to store the sum of M samples, if M is the decimation rate. If the maximum of the sample needs n bits (e.g. 18-bit adc value) and you decimate with a factor of M (e.g 1024) then you need $n+ N^2 \log(M)$ bits for the CIC filter, where N is the amount of consecutive integrators. So if you do a 4^{th} order CIC filter with a decimation rate of 1024, then you need 40 extra bits for the integrator. Starting with 18-bits samples, a signed 64-bit integer is large enough (but a double is NOT). Each stage also adds a delay of ½ M. The output is the averaged value of the previous M samples, so it represents the average over the interval of M samples. A N^{th}-order CIC filter will represent the average of about the last NM samples, so the delay with respect to the input is about NM/2.
- When you keep summing the input, the integers will overflow eventually. That is no problem as long as it does not happen more often than once every M samples. When you are close to the maximum value of the integer (2^63-1) and add 1, you get - 2^63. Subtracting these 2 numbers gives 1 again.
- Using 64-bit integers one can implement a 4-stage CIC filter with 18-bit sample input and decimation rate of 1024. The first stage is the sum of the inputs, the second the sum of the first-stage result, etc.

Cascading combs and integrators

- 4-stage CIC filter: 4 integrators in series, followed by 4 combs:



Cic filters: 4 integrators followed by 4 subtractions. Keep summing input, after N steps you have result[N].
Store this result. Subtract the stored result[N-M] to get Res2[N] (you stored this result M steps earlier in the calculation. You don't need to store or subtract results at integers that are not multiples of M).
Store this result res2[N] (to be subtracted here after N+M steps).
Subtract result res2[N-M] to get Res3[N]. store this result.
Subtract result Res3[N-M] to get res4[N].
Subtract res4[N-M] to get finally Res5.

Put Res5 in a double, divide by M^4 to normalize, and you have the output of 4 cascaded Comb-integrator filters.

CIC filters, delay

- Each integrator stage of the CIC filter averages the preceding M samples (assuming the standard setting delay=1M). Therefore, the average result at the output is delayed by M/2 samples (for instance, if your adc measures for 1 microsecond and gives a result at t_n (after n microseconds), then the CIC filter result at t_n equals the sum of the last N samples, so the integrated result from t_n – M till t_n. The central value of this interval occurs at t_n – M/2, which one could interpret as the correct time of the measurement.

- Thereby it is demonstrated that each stage in the CIC filter output introduces a delay of M/2 samples. For a 4-staged CIC filter the delay of the averaged output with respect to the input signal equals 2M samples.

CIC filter, exercise

- The CIC filter exercise is due next week monday. It is imperative that the CIC filter is implemented with integer arithmetic; doing the sums in double precision would lead to enormous errors already after 1 second, whereas in integer arithmetic the numerical error is 0.
- The exercise can be found on the web (www.nikhef.nl/~henkjan/ click on Computational Methods).
- http://www.nikhef.nl/~henkjan/NUMREC/EXERCISES/Exercise2.pdf