Floating Point and all that
A gentle reminder

Manuel Schiller

University of Glasgow

August 18th, 2022

introduction

- we all use floating point numbers on a daily basis
- we *think* we know how to calculate with them
- **but do we?**

→ maybe a gentle reminder is useful

Contents:

- floating point numbers
- basic arithmetic
- adding up many numbers
- RMS calculation
- finding zeros of functions
- useful parametrisations/approximations
- a few words on linear systems and matrix decompositions

floating point

floating point

floating point numbers

- a floating point number is basically a number in scientific notation:

$$\pm \text{mantissa} \cdot \text{base}^{\text{exponent}}$$

- on today's computers: base = 2, mantissa saved in binary (toy format for this talk)

```
+/- 1.mmmmmmm * 2^(+/-eeeee)
```

- example is with 13 bits: 7 bits for mantissa, 5 for exponent, two sign bits: `1.1010001 * 2^(+00001)` is $(1 + \frac{1}{2} + \frac{1}{8} + \frac{1}{128}) \cdot 2^1 = 3.265625$
- on real computers: IEEE 754 floating point numbers
  - single precision (`float`): 23 bits mantissa, 7 exponent bits
  - double precision (`double`): 52 bits mantissa, 10 exponent bits
  - IEEE 754 implementation detail: exponent is one bit wider than specified above and saved without sign bit, with an implied bias that has to be subtracted

floating point number classification (1/2)

- signed zero: you have $+0.0$ and -0.0
  lowest possible exponent value, mantissa zero

  `+/- 0.0000000 * 2^(-11111)`
- denormalized numbers: mantissa not normalised
  lowest possible exponent value, mantissa non-zero

  `+/- 0.0000001 * 2^(-11111)` to `+/- 0.1111111 * 2^(-11111)`
- finite normal floating point numbers
  neither lowest nor highest possible exponent value, mantissa normalised, i.e.

  `+/- 1.0000000 * 2^(-11110)` to `+/- 1.1111111 * 2^(+11110)`

floating point number classification (2/2)

- $\pm\infty$: exponent is largest possible value, mantissa zero

  `+/- 1.0000000 * 2^(+11111)`

  - larger(smaller) than largest(smallest) finite positive(negative) number
  - you "contract" it from calculations like $\frac{\pm 1}{0}$, $\log(0)$, ...
- Not a Number (NaN): exponent is largest possible value, mantissa non-zero

  `+/- 1.mmmmmmm * 2^(+11111)` with `mmmmmmm` not `0000000`

  - any calculation involving NaN yields NaN
  - usually used to indicate something wrong with inputs to calculation, e.g. $\log(-1)$, $\infty - \infty$, ...
  - two types: quiet NaN, signalling NaN (raises floating point exception)
  - not ordered: it's the only FP number unequal to itself!
- in C++: `std::fpclassify`, `std::isnan`, `std::isinf`, `std::isfinite`, `std::isnormal`

floating point arithmetic: multiplication

a=1.0110101 * 2^(00110), b=1.1001010 * 2^(00011)

1. multiply mantissas:

```
1.0110101 * 1.1001010 = 1.0110101 +
                         10110101 +
                          00000000 +
                           00000000 +
                            10110101 +
                             00000000 +
                              10110101 +
                               00000000
carry bits                 11.11110111 1
                        ---------------------
a * b =                    10.00111011010010 * 2^(00110) * 2^(00011)
```

2. add exponents

```
a * b = 10.00111011010010 * 2^(00110 + 00011)
      = 10.00111011010010 * 2^(01001)
```

3. round mantissa to correct number of bits (round-to-even!)

```
a * b = 10.001111 * 2^(01001)
```

4. normalise mantissa (bring to form 1.mmmm...), adjust exponent

```
a * b = 10.001111 * 2^(01001)
      = 1.0001111 * 2^(01010)
```

floating point arithmetic: division

- basically, you do not want to know, but...
  - in principle similar to multiplication:
    long division of mantissas, subtraction of exponents
  - on many CPUs, can use some tricks for faster calculation:
    - idea is often to use Newton's method in hardware to get $y = 1/x$:
      find zero of function $f(y) = 1/y - x$
    - lookup table to get a good approximation for a starting value
      $0 < y_0 < \frac{2}{x}$
    - then refine the answer:

$$y_{k+1} = y_k - \frac{f(y_i)}{f'(y_i)} = y_k - (y_k - x \cdot y_k) = y_k \cdot (2 - x y_k)$$

```
x = 3, y0 = 0.25 ~ 1/3 (0 < y0 < 2/3)
y0 = 0.25     dy = 0.0625 (dy = y0 - x * (y0)^2)
y1 = 0.3125   dy = 0.0195312 (dy = y1 - x * (y1)^2)
y2 = 0.332031 dy = 0.001297 (dy = y2 - x * (y2)^2)
y3 = 0.333328 dy = 5.0962e-06 (dy = y3 - x * (y3)^2)
y4 = 0.333333 dy = 0 (dy = y4 - x * (y4)^2)
```

    - then multiply: $z/x = z \cdot y$

floating point arithmetic: addition/subtraction

`a = +1.0110101 * 2^(+00001)  b = -1.0110101 * 2^(-00001)`

1. adjust exponents and add/subtract

```
a + b = +1.0110101    * 2^(+00001)
        -0.010110101  * 2^(+00001)
(borrow)     1    1
        ---------------
        +1.000100011  * 2^(+00001)
```

2. round to correct number of bits (round to even)

```
a + b = +1.000100011 * 2^(+00001)
      = +1.0110111   * 2^(+00001)
```

3. normalise exponent (if needed – here, it's okay)

```
a + b = +1.0110111 * 2^(+00001)
```

floating point: rounding

- there are four rounding modes:
  - round to nearest, ties towards even number ("round to even", default)
  - towards zero
  - towards $+\infty$
  - towards $-\infty$
- round to even: rounding to the usual 7 bits of our toy floats:
  - `1.0000000 01 = 1.0000000`
  - `1.0000000 11 = 1.0000001`
  - `1.0000000 10 = 1.0000000`
  - `1.0000001 10 = 1.0000010`
- normally, the default is fine – I have never had to change the rounding mode
- in C: `fesetround`

floating point: accuracy, machine epsilon

- if $\mathrm{rd}(x)$ is rounding for a floating point format, then there exists an ϵ for which

$$\mathrm{rd}(a \circ b) = (a \circ b)(1 + \epsilon)$$

  for $\circ \in \{+, -, \cdot, /\}$ with $|\epsilon| \leq \mathrm{eps}$
- eps depends on the floating point format (sometimes called "machine epsilon"):
  - "the smallest number you can add to one so that result is still different from one"
  - for our toy format: $\mathrm{eps} = \frac{1}{128}$
  - `std::numeric_limits<float>::epsilon()= 1.192093e-07`
  - `std::numeric_limits<double>::epsilon()= 2.2204460492503131e-16`

floating point: accuracy, special functions

- for special functions like $\sin$, $\cos$, $\exp$, …
  - the idea is similar as for addition/subtraction/multiplication/division
  - but for many of these functions, the exact result is transcendental
  - libraries cannot calculate infinitely many digits to decide how to round
  - the requirements are usually a bit relaxed
- the difference between a good math library and the exact result should be not more than $\pm$ 1 ULP
- an ULP is a unit in the last place (of the mantissa)
- "fast" math libraries are often quite a bit worse
  - check the docs
  - and/or measure
  - `std::nextafter` and `std::nexttoward` are your friends

a word on printing

- if there is even a slight chance that your output is read by another program:
  - print in scientific format (7 digits for `float`, 16 for `double`)
  - C++: `std::cout <<std::scientific <<std::setprecision(16)<<x`
  - C: `std::printf("%.16e\n", x)`
- during the run 2 commissioning, this was not done for the pickle files of the TCKs (trigger configuration keys)
  - a parameter of order 10^{-9} was printed as `0.0`
  - result was a pattern reco algorithm running with wrong parameters and not finding tracks…
- even for human consumption, please consider scientific format for everything but the final tables…

floating point: limits for float/double

- `std::numeric_limits<T>::min()/::max()` are your friends:

std::numeric_limits<T>::	T=float	T=double
+ max()	+3.402823e+38	+1.79769313486231570e+308
+ min()	+1.175494e-38	+2.22507385850720140e-308
- min()	-1.175494e-38	-2.22507385850720140e-308
- max()	-3.402823e+38	-1.79769313486231570e+308

- okay, now you know the pieces of the game...
- let's play!

common pitfalls

common pitfalls

problem: floating point addition is not what you think

- we think we know how to add numbers :

$$a + (b + c) = (a + b) + c = (a + c) + b = (b + c) + a$$

- let's use $a = 1, b = -1, c = 1 \cdot 10^{-5}$
- from our math lessons, we know that $a + b + c = 1 \cdot 10^{-5}$
- let's see if that holds true on a computer:

```
# root -l
root [0] float a= 1, b = -1, c = 1e-5f;
root [1] (a + b) + c
(float) 1.00000e-05f
root [2] a + (b + c)
(float) 1.00136e-05f
root [3] (a + c) + b
(float) 1.00136e-05f
root [4] (b + c) + a
(float) 1.00136e-05f
root [5]
```

- different answers – floating point addition is not associative!
- cancellations lose precision
- don't needlessly add/subtract numbers of different magnitudes!

warm-up: $x^2 - y^2$

- avoid taking difference of quantities of different magnitudes!
- very good (and frequent) example: $x^2 - y^2$
  - if x and y have different orders of magnitude, their squares are even more affected by this!
  - better way: $x^2 - y^2 = (x + y)(x - y)$
  - this version is just as fast (3 FLOPS), but numerically much more stable
- example: $x = 1 + 2^{-11}$, $y = 1 + 2^{-12}$, `float` precision

exact	$2^{-11}(1 + 2^{12} + 2^{-13})$
exact as `float`	`4.8846006e-04`
$(x^2 - y^2)$	`4.8851967e-04`
$(x + y)(x - y)$	`4.8846006e-04`

pairwise summation

- sum numbers in a numerically relative safe way
  - (sort numbers by ascending $|x_k|$)
  - sum pairs of numbers
  - then sum pairs of the result of summing pairs, and so on...
- for N numbers, each participates only in $\mathcal{O}(\log N)$ additions (FFT algorithms tend to not have issues for the same reason)
- be careful, routine below changes the input array!

```
float safe_sum(std::vector<float>& arr)
{
    if (arr.begin() = last) return 0.f;
    std::sort(arr.begin(), arr.end(), [] (const auto& a, const auto& b) {
        return std::abs(a) < std::abs(b);});
    auto last = arr.end();
    // while there are two or more numbers in arr, we still have work to do
    while (arr.begin() + 1 != last) {
        auto newlast = arr.begin();
        // add numbers in pairs, save results from beginning of arr
        for (auto it = newlast; last != it; ++newlast) {
            const auto pairsum = *it++;
            if (last != it) pairsum += *it++;
            *newlast = pairsum;
        }
        // arr now has only half the size, so update the pointer to last valid
        // element
        last = newlast;
    }
    return arr.front();
}
```

Kahan summation

- if the previous routine does not get you out of trouble yet:
- →Kahan summation works by
  - accumulating of the error introduced by truncation
  - periodically fold back the accumulated truncation error into the sum
- basically another trick to postpone numerical trouble

```
template <class T>
typename T kahan_sum(const std::vector<T>& arr)
{
    T sum = 0; // sum
    T c = 0;   // keep track of lost digits
    for (const auto& el: arr) {
        const T y = el - c;  // fold back lost digits
        const T t = sum + y; // new value of sum
        c = (t - sum) - y;   // work out digits lost in sum
        sum = t;             // update sum
    }
    return sum;
}
```

calculating variance

calculating variance

calculating variances: bad algorithm

- in statistics books, you often find this formula

$$\mathrm{Var}(x) = \frac{\left(\sum_{k=1}^{N} x_k^2\right) - N\left(\sum_{k=1}^{N} x_k\right)^2}{N-1} = \frac{\left(\sum_{k=1}^{N} x_k^2\right) - N\overline{x}^2}{N-1}$$

- consequently, the code one sees most often is something like this:

```
float variance_ugly(const std::vector<float>& data)
{
    float sx = 0, sx2 = 0;
    for (const auto& x: data) {
        sx  += x;
        sx2  += x * x;
    }
    const auto n = data.size();
    return (sx2 - n * sx * sx) / (n - 1);
}
```

- the math is 100% correct, but this code goes wrong numerically
- sx2 and sx * sx have needlessly different magnitudes!
- using `double` just postpones the problem, but it'll go wrong, too!
- how do we fix this?

calculating variances: good algorithm

- an equivalent, but numerically much better formulation is this:

$$\mathrm{Var}(x) = \frac{\sum_{k=1}^{N} (x_k - \overline{x})^2}{N-1}$$

- the code is straightforward:

```
float variance_good(const std::vector<float>& data)
{
    float savg = 0;
    for (const auto& x: data) savg += x;
    const auto n = data.size();
    savg /= n;
    float svar = 0;
    for (const auto& x: data) {
        const auto diff = x - savg;
        svar += diff * diff;
    }
    return svar / (n - 1);
}
```

aka two-pass algorithm

- numerically much more stable: cancellation happens in the calculation of `diff` (same order of magnitude!)
- this should be your go-to algorithm!
- downside: need two passes (loops) over data

calculating variances: better algorithm

- an equivalent, but numerically much better formulation is this:

$$\mathrm{Var}(x) = \frac{\sum_{k=1}^{N}(x_k - \overline{x})^2 - \frac{1}{N}\left(\sum_{k=1}^{N}(x_k - \overline{x})\right)^2}{N-1}$$

- second term corrects roundoff in the first term
- the code is still fairly straightforward:

```
float variance_better(const std::vector<float>& data)
{
    float savg = 0;
    for (const auto& x: data) savg += x;
    const auto n = data.size();
    savg /= n;
    float svar = 0, scorr = 0;
    for (const auto& x: data) {
        const auto diff = x - savg;
        scorr += diff;
        svar += diff * diff;
    }
    return (svar - scorr * scorr / n) / (n - 1);
}
```

aka →corrected two-pass algorithm

- downside: still need two passes (loops) over data

calculating variances: best algorithm

- in principle, this version uses same math as previous two methods
- but: keep running averages (use definitions of mean and variance to get recurrence relations and code below)

$$\delta_k = x_k - \overline{x}_{k-1}$$
$$\overline{x}_k = \overline{x}_{k-1} + \frac{\delta_i}{k}$$
$$v_k = v_{k-1} + (x_k - \overline{x}_k)\delta_k$$

- the code is still fairly straightforward:

```
float variance_best(const std::vector<float>& data)
{
    float mean = 0, var = 0;
    std::size_t n = 0;
    for (const auto& el : arr) {
        const auto delta = el - mean;
        mean += delta / (n + 1);
        var += (el - mean) * delta;
        ++n;
    }
    return var / (n - 1);
}
```

aka →Welford's Online Algorithm

- only single pass over data required

results

- let's compare using toy data: $2^{20} \sim 1$M Gaussian random numbers
- calculate standard deviation (square root of variance) for $\mu = 1, \sigma = 10^{-2}$

algorithm	ugly	good	better	best
`float`	1.2241169e-02	1.0005541e-02	1.0005539e-02	1.0005564e-02
`double`	1.0008116e-02	1.0008124e-02	1.0008122e-02	1.0008122e-02

- calculate standard deviation (square root of variance) for $\mu = 1, \sigma = 10^{-4}$

algorithm	ugly	good	better	best
`float`	3.4526715e-04	9.9996367e-05	9.9996330e-05	9.9997254e-05
`double`	5.8856527e-05	1.0002454e-04	1.0002450e-04	1.0002450e-04

- as you can see, calculating in `double` is not enough
- you need a good algorithm, too!

weighted data

- generalisation to weighted data straightforward

```
float variance_good(const std::vector<float>& data
                    const std::vector<float>& weights)
{
    float savg = 0, wsum = 0;
    auto it = weights.cbegin();
    for (const auto& x: data) {
        const auto& w = *it++;
        wsum += w;
        savg += w * x;
    }
    savg /= wsum;
    float svar = 0;
    it = weights.cbegin();
    for (const auto& x: data) {
        const auto& w = *it++;
        const auto diff = x - savg;
        svar += w * diff * diff;
    }
    return svar / (wsum - 1);
}
```

```
float variance_best(const std::vector<float>& data,
                    const std::vector<float>& weights)
{
    float mean = 0, var = 0, wsum = 0;
    auto it = weights.cbegin();
    for (const auto& el : arr) {
        const auto& w = *it++;
        wsum += w;
        const auto delta = el - mean;
        mean += (w / wsum) * delta;
        var += w * (el - mean) * delta;
    }
    return var / (wsum - 1);
}
```

libraries

- Which algorithm does your favourite library use?
  - ROOT's `TH1` classes: ugly algorithm in `double` (→link)
  - ROOT's `RH1` classes: same (I think – code is hard to read – →link)
  - RooFit's `RooAbsData`: good algorithm + Kahan summation (→link)
  - NumPy and Pandas: good algorithm (→link and →link)
  - boost's histogram: best algorithm (→link)
- check your favourite library!
  - if it's ugly, consider fixing it
  - ROOT's histogram classes not easily fixable due to large number of persisted histograms
- looked at some of the obvious places, but if you use something, you need to check what's used inside

zeros of functions

zeros of functions

zeros of functions

- frequent task: find x for which $f(x) = 0$
- two (common) solutions:
  - bisection: if you know that your zero is between a and b

```cpp
// #include <cmath>, #include <cstdio>, #include <limits>

static float f(float x) { return x * x - 2.f; }

static float bisect(float (*f)(float), float a, float b, float xtol, float ftol)
{
    float fa = f(a), fb = f(b);
    for (unsigned nmax = 100; nmax; --nmax) { // avoid getting stuck
        const float mid = 0.5f * (a + b);
        const float fmid = f(mid);
        // look for interval in which f changes sign
        if (fa * fmid <= 0.f) b = mid, fb = fmid;
        else if (fb * fmid <= 0.f) a = mid, fa = fmid;
        if (std::abs(b - a) <= xtol || std::abs(fmid) <= ftol) return mid;
    }
    return std::numeric_limits<float>::quiet_NaN();
}

int main()
{
    const float eps = std::numeric_limits<float>::epsilon();
    std::printf("sqrt(2) from bisection: %e\n", bisect(f, 1.f, 2.f, 2.f * eps, 2.f * eps));
    return 0;
}
```

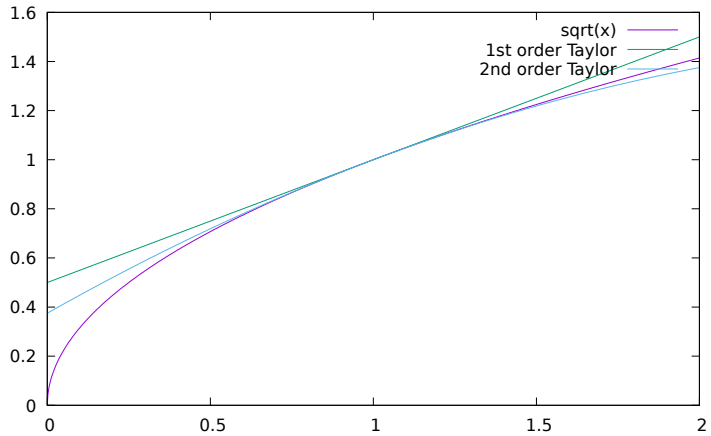
  - Newton's method (see example with floating point division)

approximation

approximation

approximating functions

- often, we have to approximate/parametrise functions
- we all know about Taylor expansions and lookup tables
- Taylor expansions don't "fit well at the edges"
- lookup tables consume memory



- often better: Chebyshev approximation

Chebyshev polynomials

- definition: $(x \in [-1,1])$

$$T_0(x) = 1 \qquad T_1(x) = x \qquad T_n = 2xT_{n-1}(x) - T_{n-2}(x)$$

or

$$T_n(x) = \cos(n \arccos(x))$$

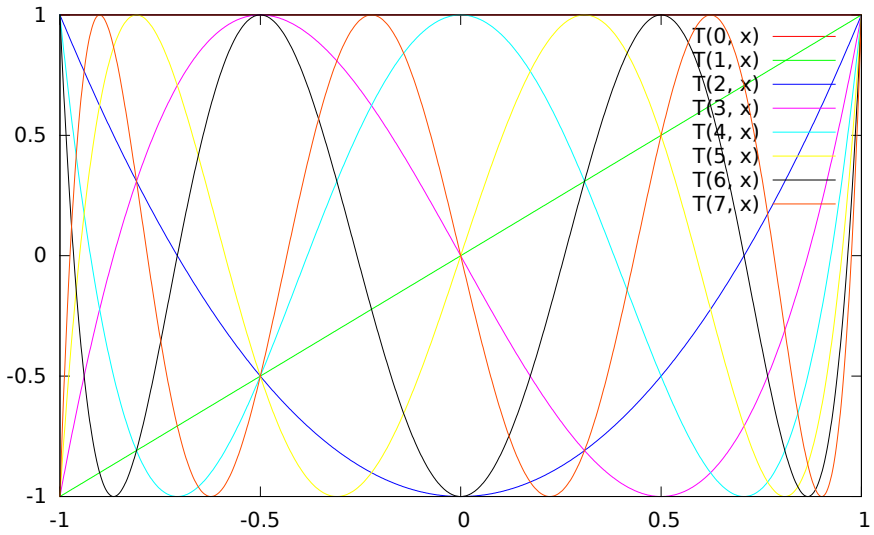
- these are orthogonal:

$$\int_{-1}^{1} T_j(x)T_k(x)\frac{dx}{\sqrt{1-x^2}} = \begin{cases} 0 & j \neq k \\ \frac{\pi}{2} & j = k \neq 0 \\ \pi & j = k = 0 \end{cases}$$

or

$$\sum_{l=0}^{N} T_j(x_l)T_k(x_l) = \begin{cases} 0 & j \neq k \\ N & j = k = 0 \\ \frac{N}{2} & j = k \neq 0 \end{cases} \qquad (x_l = \cos(\frac{l\pi}{N}))$$

Chebyshev polynomials



- approximate $f(x) \approx \sum_{k=0}^{n} c_k T_k(x)$
- Chebyshev polynomials intimately related with Fourier transforms
  - → fast convergence for well behaved functions
- best of all: error estimates are easy: $|T_k(x)| \leq 1$
  - → accurate error estimate from summing up first few neglected $|c_k|$

Chebyshev-expanded walk relation for LHCb OT

- in runs 1 and 2, LHCb has Outer Tracker (drift tube detector) behind dipole
- measures drift time of ions in counting gas to get position
- ionisation signal attenuated as it travels along 2.5 m long anode wire

→ weaker signal means drift time measurement is late: walk relation

- have to approximate for $l \in [0, 3600]$ mm, only use approx. in pattern reco
- truncate after c_4 (e.g. with parameter's from Alexandr's thesis):

$c_0 = -0.116724, c_1 = 0.544860, c_2 = -0.290254, c_3 = 0.196250, c_4 = -0.110068$

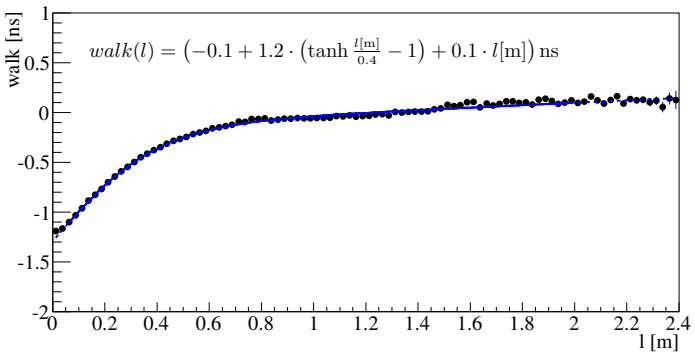
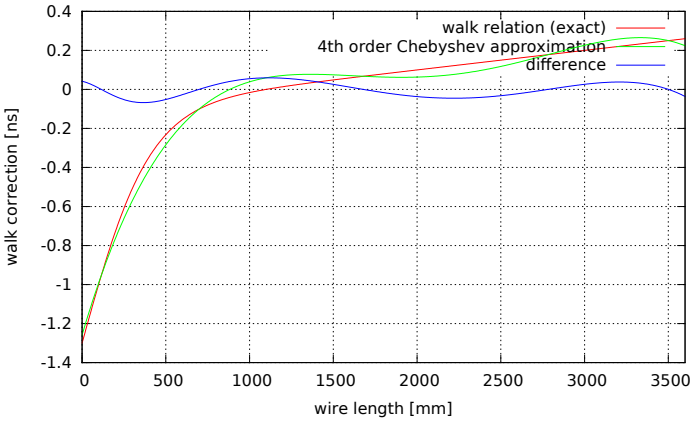


Figure 4.15: Average drift time residual versus the distance to the straw end along the wire. The line shows the walk parameterization fit.

Alexandr Vladimirovich Kozlinskiy's PhD thesis, Amsterdam, 2013



est. error: 0.0788 ns (scanning shows max. deviation < 0.0676 ns)

conclusion

- at one time, tanh evaluation was 30% of HLT CPU, Chebyshev completely negligible
- no impact on reconstruction performance
- Chebyshev expansion very useful tool!
  - fast approximations can be engineered to accuracy requirements
  - code available in package Kernel/LHCbMath
  - it's even easy to use:

```
#include "LHCbMath/ChebyshevApprox.h"
ChebyshevApprox<float, 5> walkrel(0., 3600.,
        [] (double l) { return -0.1 + 1.2 * (std::tanh(l / 400.) - 1.)
              + 1e-4 * l; });

double walkcorr = walkrel(125.);
```

- you can also use it to integrate: see RooChebyshev.cxx in ROOT sources

linear systems

linear systems

introduction

- task: given matrix A and vector $\vec{b}$, find $\vec{x}$ such that $A\vec{x} = \vec{b}$
- easy if A is triangular
  - can solve component by component by substitution:

$$\begin{pmatrix} a_{11} & 0 & \dots & 0 \\ a_{21} & a_{22} & \ddots & \vdots \\ \vdots & & \ddots & 0 \\ a_{n1} & \dots & \dots & a_{nn} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

  - solving for a single vector faster than inverting matrix
  - floating point is not exact, so there is always roundoff
- what if matrix is not triangular?

choosing matrix decompositions

- decompose matrix, e.g. $A = LU$ where decomposed matrices are easy to solve/invert/...
- works by applying series of similarity transforms T_i to original matrix
- available in many libraries
- choice of T_i governs how much roundoff is amplified
- condition number $\kappa(T_i) = \frac{\text{largest eigenvalue of } T_i \text{ in absolute value}}{\text{smallest eigenvalue of } T_i \text{ in absolute value}}$
- different decomposition methods have different tradeoffs:
  - LU decomposition (fast): $\kappa(T_i) \geq 1$, sometimes much larger
  - A symmetric, positive definite (fits!): Cholesky decomposition ($\kappa = 1$)
  - A invertible: QR decomposition ($\kappa = 1$)
  - A difficult: Singular Value Decomposition (SVD) ($\kappa = 1$, A singular allowed)

summary

summary

summary

- floating point is hard!
- the naïve solution is not necessarily safe
- hope everybody now knows how to calculate sums/variance/RMS/… safely!
- code is available for you to play with
- further reading: see backup

Thanks, and I hope it was interesting!

backup

backup

further reading

- W.H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery: Numerical recipes in C - The Art of Scientific Computing, 2nd edition, Cambridge University Press, subsection 14.1
- →Wikipedia article about calculating variance