Floating Point in Experimental HEP Data Processing

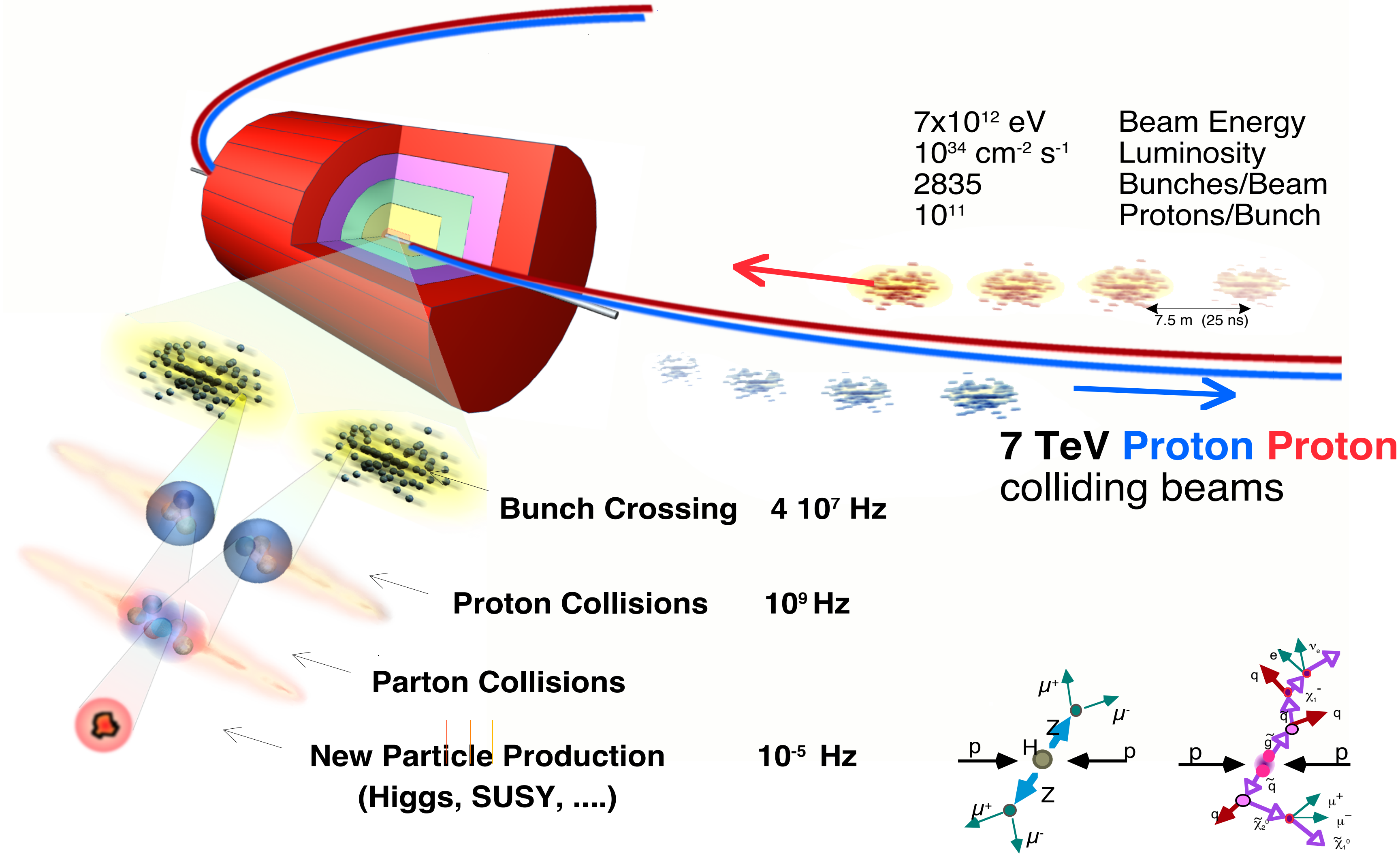
(aka Reconstruction)

Vincenzo Innocente

CERN

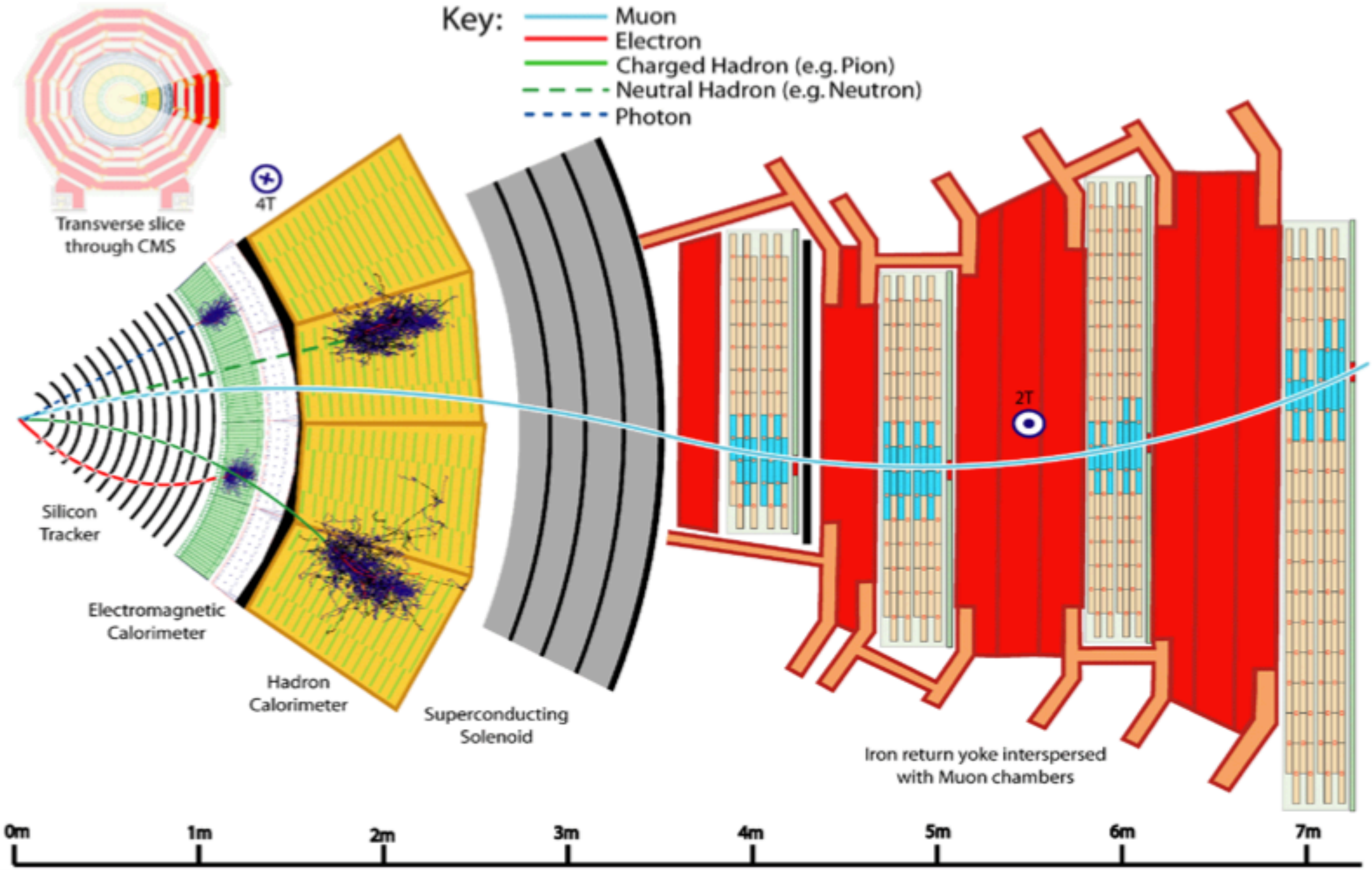
PH/SFT & CMS

Collisions at the LHC: summary

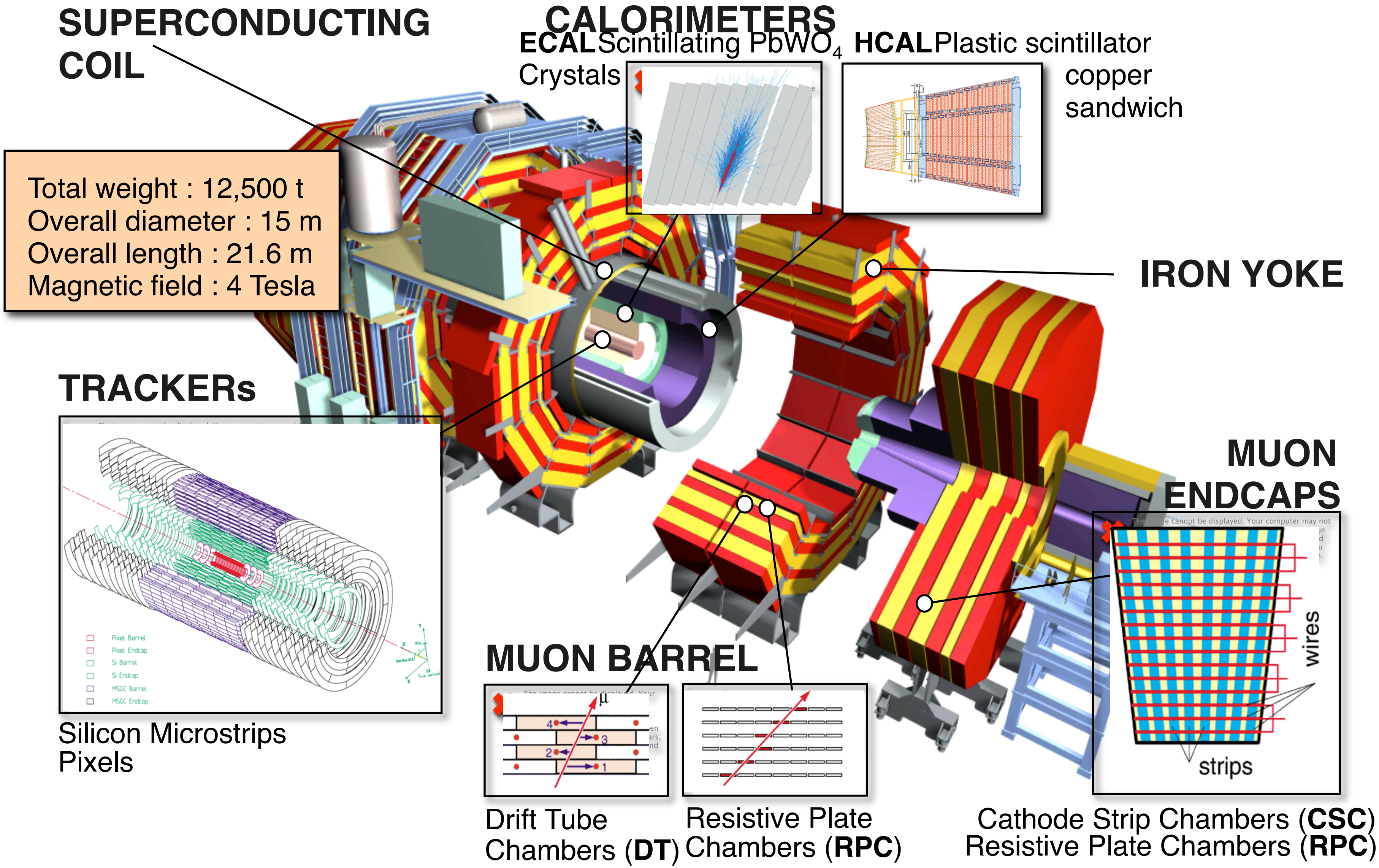


Selection of 1 event in 10,000,000,000,000

Detector "onion" structure



An experiment: CMS

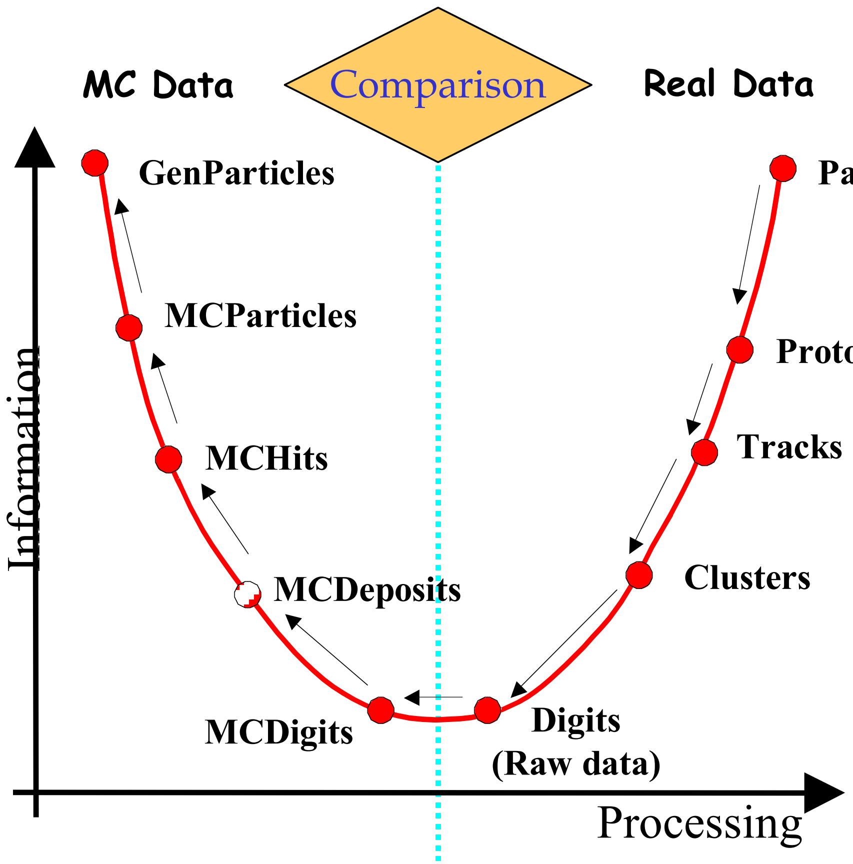


Data and Algorithms

- HEP main data are organized in *Events* (particle collisions)
- Simulation, Reconstruction and Analysis programs process "one Event at the time"
  - Events are fairly independent of each other
  - Trivial parallel processing
- Event processing programs are composed of a number of Algorithms selecting and transforming "raw" Event data into "processed" (reconstructed) Event data and statistics
  - Algorithms are mainly developed by "Physicists"
  - Algorithms may require additional "detector conditions" data (e.g. calibrations, geometry, environmental parameters, etc.)
  - Statistical data (histograms, distributions, etc.) are typically the final data processing results

High Energy Analysis Model

MonteCarlo Simulation follows the evolution of physics processes from collision to digital signals



Reconstruction "goes back in time" from digital signals to the original particles produced in the collision

Analysis compares (at statistical level) reconstructed events from real data with those from simulation

Analogies with Industry

- Signal/image processing
  - DAC (including calibrations)
  - Pattern recognition, "clustering"
- Topological problems
  - Closest neighbor, minimum path, space partitioning
- Gaming *(our main source of inspiration!)*
  - "walk-through" complex 3D geometries
  - Detection of "collisions"
- Navigation/Avionics (Kalman filtering)
  - Tracking in a force field in presence of "noise"
  - Trajectory identification and prediction

Accuracy, Precision

- Measurement themselves require a modest precision (16,24 bits)
- Geometry/Materials often known at per-cent level
- Dynamic range, when converted in natural units, often requires a high precision FP representation
  - Enengy range $>10^9$
  - Position: micron over 20m
- Many conversions back and forth various coordinate/measurement systems
- Error manipulation (including correlations)
  - Squared quantities: each transformation requires two matrix multiplications

FP operations in reconstruction

- Signal calibration
  - Ideal for vectorization
    - (if was not that calib requires lookup!)
    - Calib-params may depend on “reconstructed quantities”
- “Geometry” transformation
  - Trigonometry (also log/exp!)
  - Small matrices (max 5x5, 6x6)
- Many logs, exp coming from parameterizations

Vectorization?

- Current code design and implementation often hinder vectorization
  - High granularity "naïve" object model
    - Innermost loop often not the longest!
  - Fragmentation in several libraries (plugin model)
    - Ito will not help
  - "Linear thinking" conditional code
- Only a massive redesign of data-structures and algorithms will make vectorization effective
  - Not alone: see
    - http://research.scee.net/files/presentations/gcapaustralia09/Pitfalls_of_Object_Oriented_Programming_GCAP_09.pdf
    - http://www.slideshare.net/DICEStudio/introduction-to-data-oriented-design

Typical Profile (today)

CPI (cycle per instruction): 0.964
load instructions %: 30.58%
store instructions %: 13.74%
load and store instructions %: 44.31%
resource stalls % (of cycles): 30.63%
branch instructions % (approx): 17.06%
% of branch instr. mispredicted: 2.25%
% of L3 loads missed: 2.09%
computational x87 instr. %: 0.038%
Div. Busy: 5.30%

% of SIMD in all uops: 19.22%
% of comp. SIMD in all uops: 10.17%

breakdown:	%of all uops	% of all SIMD
PACKED_DOUBLE:	0.663%	3.449%
PACKED_SINGLE:	0.613%	3.190%
SCALAR_DOUBLE:	13.485%	**70.159**%
SCALAR_SINGLE:	4.038%	21.010%
VECTOR_INTEGER:	0.421%	2.192%

More details (see next page):
Function where time is spent most

- *No hot-spot: top 30 each between 2.5% and 0.5% of total*
- Trig/trans functions
- div/sqrt latency

BR_INST_EXEC.INDIRECT_NON_CALL		UOPS_RETIRED.STALL_CYCLES		ARITH.CYCLES_DIV_BUSY		Function
9.5e+07	5.30 %	8.1e+09	41.41 %	2e+09	10.07 %	__ieee754_exp
3.5e+08	13.71 %	8.1e+09	45.49 %	0	0.00 %	arena_malloc_small
6.7e+06	0.23 %	7.5e+09	47.55 %	3.8e+09	24.31 %	__ieee754_atan2
6.6e+07	46.92 %	9.9e+09	63.11 %	4.2e+09	26.82 %	void TkGluedMeasurementDet::doubleMatch< ...
1.9e+08	15.15 %	4.9e+09	33.67 %	0	0.00 %	arena_dalloc_bin
1.4e+08	7.66 %	9.6e+09	68.94 %	5.9e+09	42.28 %	ThirdHitPredictionFromCircle::phi(double ...
3.4e+07	1.05 %	6e+09	43.11 %	3.6e+09	25.47 %	atanf
3.9e+08	17.85 %	7.8e+09	58.89 %	0	0.00 %	free
4.4e+07	2.68 %	8.5e+09	65.22 %	2.4e+09	18.60 %	__ieee754_acos
2.5e+07	2.56 %	4.3e+09	34.11 %	1.1e+08	0.90 %	ROOT::Math::SMatrix<double, (unsigned in ...
1.1e+07	11.71 %	4.4e+09	41.21 %	0	0.00 %	cms::TrackListMerger::produce(edm::Event ...
8.5e+07	204.00 %	8.6e+09	81.25 %	4.2e+09	39.96 %	magfieldparam::TkBfield::Bcyl(double, do ...
6.2e+06	0.59 %	4.6e+09	46.46 %	5.6e+08	5.70 %	__ieee754_log
1.7e+06	0.99 %	4.9e+09	53.99 %	5.6e+07	0.61 %	<unknown(s)>
1.8e+08	7.49 %	5.1e+09	59.85 %	2.8e+07	0.33 %	strcmp
2.6e+08	20.20 %	5.5e+09	67.64 %	2.6e+09	32.26 %	PixelTripletLargeTipGenerator::hitTriple ...
0	0.00 %	4.3e+09	57.80 %	1.1e+08	1.51 %	do_lookup_x
9.3e+07	11.99 %	4.9e+09	66.54 %	3.9e+09	53.23 %	DAClusterizerInZ::update(double, std::ve ...
3.4e+07	11.88 %	3.5e+09	48.00 %	3.1e+08	4.22 %	sincos
1.3e+08	24.73 %	2.5e+09	41.40 %	4.2e+08	6.82 %	PixelTripletHLTGenerator::hitTriplets(Tr ...
4.8e+07	19.87 %	4.7e+09	77.57 %	4.5e+08	7.34 %	tan
0	0.00 %	2.5e+09	45.01 %	0	0.00 %	<unknown(s)>
7.3e+07	8.77 %	2.1e+09	37.74 %	5.9e+08	10.71 %	__ieee754_atan2f
9.8e+06	5.74 %	3.9e+09	71.26 %	2e+09	37.42 %	AnalyticalCurvilinearJacobian::computeFu ...
8.4e+06	9.26 %	3.4e+09	64.46 %	1.5e+09	28.77 %	JacobianCurvilinearToLocal::JacobianCurv ...
7.3e+06	9.85 %	1.7e+09	32.66 %	0	0.00 %	SiStripRecHit2D::sharesInput(TrackingRec ...
6.7e+07	24.80 %	3.1e+09	62.12 %	1.2e+09	23.72 %	StripCPEfromTrackAngle::localParameters(...
2.4e+07	17.47 %	2.9e+09	62.58 %	7e+08	15.34 %	std::pair<bool, double> Chi2MeasurementE ...
1.6e+08	13.06 %	1.7e+09	36.84 %	0	0.00 %	arena_malloc
0	0.09 %	5.3e+08	12.62 %	0	0.00 %	PixelHitMatcher::compatibleSeeds(std::ve ...
6.6e+07	23.53 %	2.9e+09	69.80 %	2e+09	47.86 %	ThirdHitPredictionFromCircle::angle(doub ...
2.8e+05	5.50 %	1.8e+09	43.09 %	1.7e+09	41.04 %	RectangularPlaneBounds::inside(Point3DBa ...
2.8e+05	0.04 %	1.1e+09	28.79 %	0	0.00 %	inflate_fast
0	0.00 %	2.3e+09	59.12 %	0	0.00 %	fesetenv

Cost of operations (in cpu cycles)

op	instruction	sse s	sse d	avx s	avx d
+,-	ADD,SUB	3	3	3	3
== < >	COMISS CMP..	2,3	2,3	2,3	2,3
f=d d=f	CVT..	3	3	4	4
|,&,^	AND,OR	1	1	1	1
*	MUL	5	5	5	5
/,sqrt	**DIV, SQRT**	**10-14**	**10-22**	**21-29**	**21-45**
1.f/ , 1.f/sqrt	RCP, RSQRT	5		7	
=	MOV	1,3,...	1,3,...	1,4,....	1,4,...

Cost of functions (in cpu cycles i7sb)

	Gnu libm s	d	Cephes scalar s	d	Cephes autovect s	d	Cephes handvect s	Approx (16bits)	Intel svml s	d	Amd libm s	d
sin,cos large x	55 ***>500***	***390***	30	50	11	30	20		12	30	25	45
sincos	80		40		15		22				50	
atan2	54	120	30		13				17	52	67	87
exp	47	***370***	42	55	10	23	27		12	26	16	36
log	57	120	37	42	11	28	24	12	12	30	27	59

feraiseexcept

Where/how can we improve?
1) std math lib

- Cost of a sin/cos/exp close to div/sqrt and to the overhead of an indirect function call
  - **Inline math functions**
    - Help autovectorization too
- math-funs spend not negligible time in range reductions and limit/exceptions checking/setting
  - Our angles are ALL in [-pi,pi] range (sometime less)
  - Arguments of log/exp often in a limited range
    - **Special version for reduced ranges**

Where/how can we improve?
2) Precision, accuracy

- Double precision often required to keep under control coordinate system transformations (in particular for the error matrices)
  – Develop more robust algorithms
  – avoid back&forth
  – Choose (dynamically?) units (metrics) to avoid too large dynamic-ranges
- Tune precision to the required accuracy in parameterization
  – Use a math-lib allowing control of precision
- rsqrt/rcp (+ "tunable" Newton-Raphson)
  – C-implementation in double precision faster than sse!

Example: multiple scattering

```
double ms(double radLen, double m2, double p2) {
  constexpr double amscon = 1.8496e-4;    // (13.6MeV)**2
  double e2      = p2 + m2;
  double beta2  = p2/e2;
  double fact = 1.f + 0.038f*log(radLen);  fact *=fact;
  double a = fact/(beta2*p2);
  return amscon*radLen*a;
}
```

Already an approximation

Material density, thickness, track angle Known at percent?

```
float msf(float radLen, float m2, float p2) {
  constexpr float amscon = 1.8496e-4;    // (13.6MeV)**2
  float e2      = p2 + m2;

  float fact = 1.f + 0.038f*dirtylogf<2>(radLen); fact /= p2;
  fact *=fact;
  float a = e2*fact;
  return amscon*radLen*a;
}
```

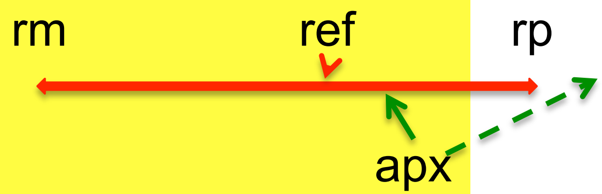
2nd order polynomial by FdD

Verify accuracy of approximation

```
float ref = ms(rl,m2,p2);
float rp = ms(rl*1.001,m2,p2);   // 0.1% positive
float rm = ms(rl*0.999,m2,p2);  // 0.1% negative
float apx = msf(rl,m2,p2);   // fast approximation

// look if approximation inside uncertainty-interval
int dd = std::min(abs(diff(rm,ref)),abs(diff(rp,ref)));
dd -= abs(diff(apx,ref)); // negative if apx-ref is larger than the uncer-interval
dm = std::min(dm,dd);

da = std::max(da,abs(diff(apx,ref)));  // maximum "error" by approx
di = std::max(di,abs(diff(rp,ref)));
di = std::max(di,abs(diff(rm,ref)));    // maximum uncertantly
// ditto for minimum
```



diff is in "bits"

- 0.1% accuracy corresponds to a difference of 13-14 bits
- Maximum error of the approximation is ~12 bits
- "dm" always positive

Cash-Karp Runge-Kutta Step

3. A STRATEGY FOR DEALING WITH NONSMOOTH BEHAVIOR

The Runge-Kutta formula derived in the previous section has the special property that it contains imbedded solutions of all orders less than five. In addition, the formula has been designed so that the first five c_i values span the range [0, 1] with reasonable uniformity, so that we have a very good chance of spotting bad behavior in f if it occurs. Our aim is to derive an automatic strategy that allows us to quit early, i.e., before all six function evaluations have been computed on the current step, if we suspect trouble, and to accept a lower order solution if appropriate.

We assume that we have computed a numerical solution y_{n-1} at the step point x_{n-1} and that for the current step, from x_{n-1} to $x_n = x_{n-1} + h$, all six function evaluations are computed so that solutions of all orders from 1 to 5 are available. (We guarantee this situation for the first step with $n = 1$). We denote the imbedded solution of order i at x_n by $y_n^{(i)}$, $1 \le i \le 5$, and define

$$\mathrm{ERR}(n, i) = \| y_n^{(i+1)} - y_n^{(i)} \|^{1/(i+1)}, \quad \text{for} \quad i \in 1, 2, 4. \qquad (6)$$

We exclude the case $i = 3$ for two reasons. First, following the approach of Shampine et al. [15], we allow only a few different orders to be used, and we have chosen to allow orders 2, 3, or 5. Second, ERR(n, 3) is of no use in predicting when to quit early since all six k_i's are required before $y_n^{(4)}$ can be computed.

Suppose now that we were to accept the solution of order 5 at x_n. We wish to compute a suitable step length, $\bar{h}_4$, to be used in integrating from x_n to x_{n+1} using a 5(4) formula. A typical step-choosing strategy would compute $\bar{h}_4$ as

$$\bar{h}_4 = \frac{\mathrm{SF} \times h}{\mathrm{E}(n, 4)}, \quad \text{where} \quad \mathrm{E}(n, 4) = \frac{\mathrm{ERR}(n, 4)}{\epsilon^{1/5}}. \qquad (7)$$

Here ϵ is the local accuracy required (as specified by the user) and SF is a safety factor often taken to be 0.9. Similarly, if we were to accept either the second- or third-order solution at x_n, the steplengths $\bar{h}_1$, $\bar{h}_2$, respectively, that would be selected at the next step by our step-control algorithm would be

$$\bar{h}_i = \frac{\mathrm{SF} \times h}{\mathrm{E}(n, i)}, \quad \text{where} \quad \mathrm{E}(n, i) = \frac{\mathrm{ERR}(n, i)}{\epsilon^{1/(i+1)}}, \quad i \in 1, 2. \qquad (8)$$

ACM Transactions on Mathematical Software, Vol. 16, No. 3, September 1990.

0.9*step/pow(err/eps,0.2)

```
// From http://martin.ankerl.com/2012/01/25/optimized-approximative-pow-in-c-and-cpp/
// good at 10%
inline double fastPow(double a, double b) {
   union { double d; int x[2]; } u = { a };
   u.x[1] = (int)(b * (u.x[1] - 1072632447)
                        + 1072632447);
   u.x[0] = 0;
   return u.d;
}
```

One More example

- Vavilov distribution is used for precise modeling of energy loss
- In CMS it is used to compute the probability of a cluster in a Silicon Detector to come from a minimum ionizing particle

$$f(\Delta, s)\, d\Delta = \frac{1}{\xi}\, \varphi_v(\lambda_v, \kappa, \beta^2)\, d\lambda_v ,$$

where

$$\varphi_v(\lambda_v, \kappa, \beta^2) = \frac{1}{\pi}\, e^{\kappa(1+\beta^2\gamma)} \int_0^{\infty} e^{\kappa f_1} \cos(y\lambda_v + \kappa f_2)\, dy$$

$$\lambda_v = \frac{\Delta - \bar{\Delta}}{\epsilon_{max}} - \kappa(1+\beta^2-\gamma)$$

$$\gamma = 0.577216 \text{ --- (Euler's constant)}$$

$$f_1(y) = \beta^2[\log y + Ci(y)] - \cos y - y\, Si(y)$$

$$f_2(y) = y\,[\log y + Ci(y)] + \sin y + \beta^2 Si(y)$$

$$Si(y) = \int_0^y \frac{\sin u}{u}\, du \text{ (sine integral)}$$

$$Ci(y) = \int_{-\infty}^y \frac{\cos u}{u}\, du \text{ (cosine integral)} .$$

- It is then encoded in an 8-bit quality word
- Precision tuned-down while verifying that the final result (the 8-bits!) do not change
  - Speed up of a factor 3…
  - More is surely possible

Modernization!

- Many algorithms coded in the `80 (even `70)
- Programmer's heuristics still based on x87 math and sequential processing
- Advent of "extreme" architectures (GPUs etc) is an opportunity to modernize algorithms for ALL architectures!

Title of program: VAVILOV

Catalogue number: AAUJ

Program obtainable from: CPC Program Library, Queen's University of Belfast, N. Ireland (see application form in this issue)

Computer: CDC 6600; *Installation*: CERN, Geneva

Operating system: CDC Scope

Programming language used: FORTRAN IV

High speed storage required: 3246 words

No. of bits in a word: 60

Overlay structure: None

No. of magnetic tapes required: None

Other peripherals used: Card reader, pine printer

No. of cards in combined program and test deck: 636

Card punching code: BCD

Keywords: Nuclear, Vavilov distribution, energy loss, thin absorber, random number generation.

Summary

- FP accounts for ~20% of HEP reconstruction
  - Mostly double (for no good reason?)
  - Not easy to vectorize as it stands
  - Large use of std math-function
- glibm: excellent full-precision reference
  - An overkill for any practical application
- Opportunities for improvements
  - Move to Data-oriented-Design
  - Reduce branches and indirect-calls
  - Use polynomial Parameterization also for non-elementary functions
  - Use fast (less precise, limited-range) math-fun
  - Use metrics that will allow the use of floats
  - Systematically verify required accuracy