Numerical analysis of MG5 CUDACPP

11 Feb. 2026, MadGraph General Meeting 2026,
Heidelberg

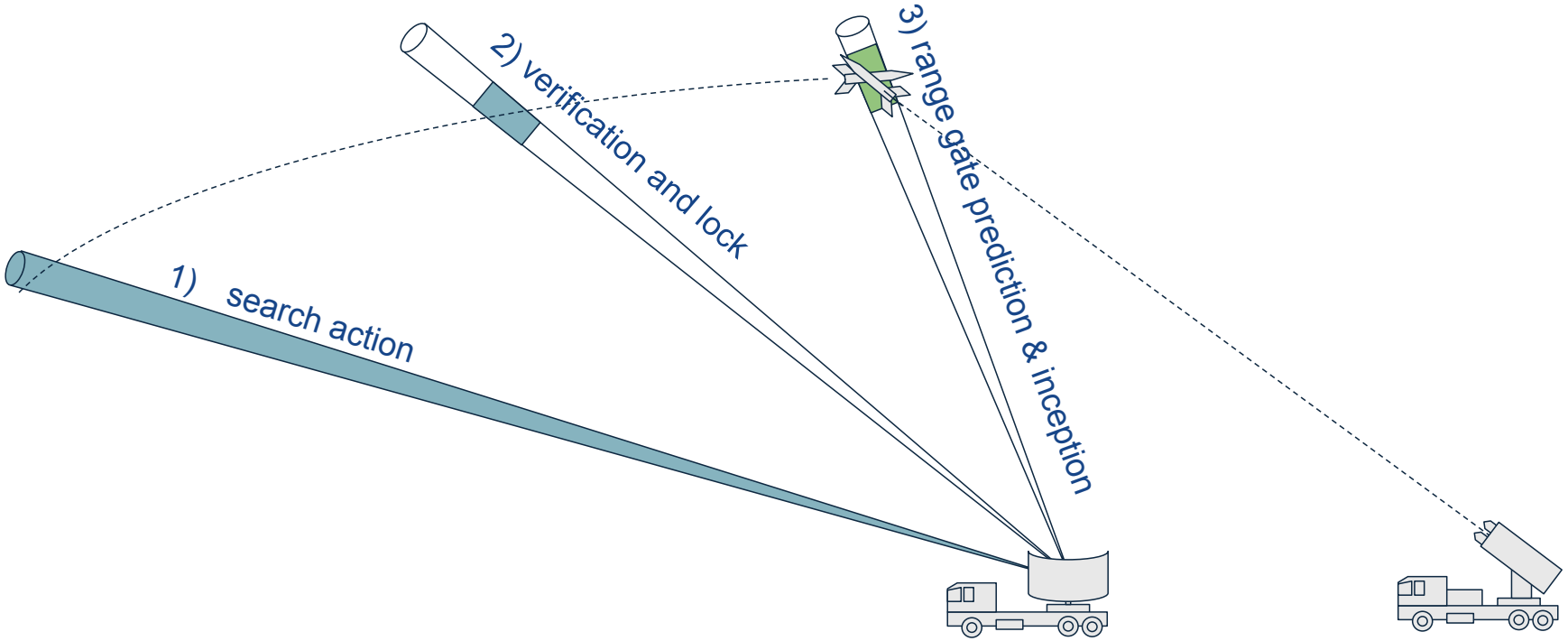
František Stloukal
CERN, LIP6 Sorbonne

NexTGen
Next Generation Triggers

The Patriot Missile

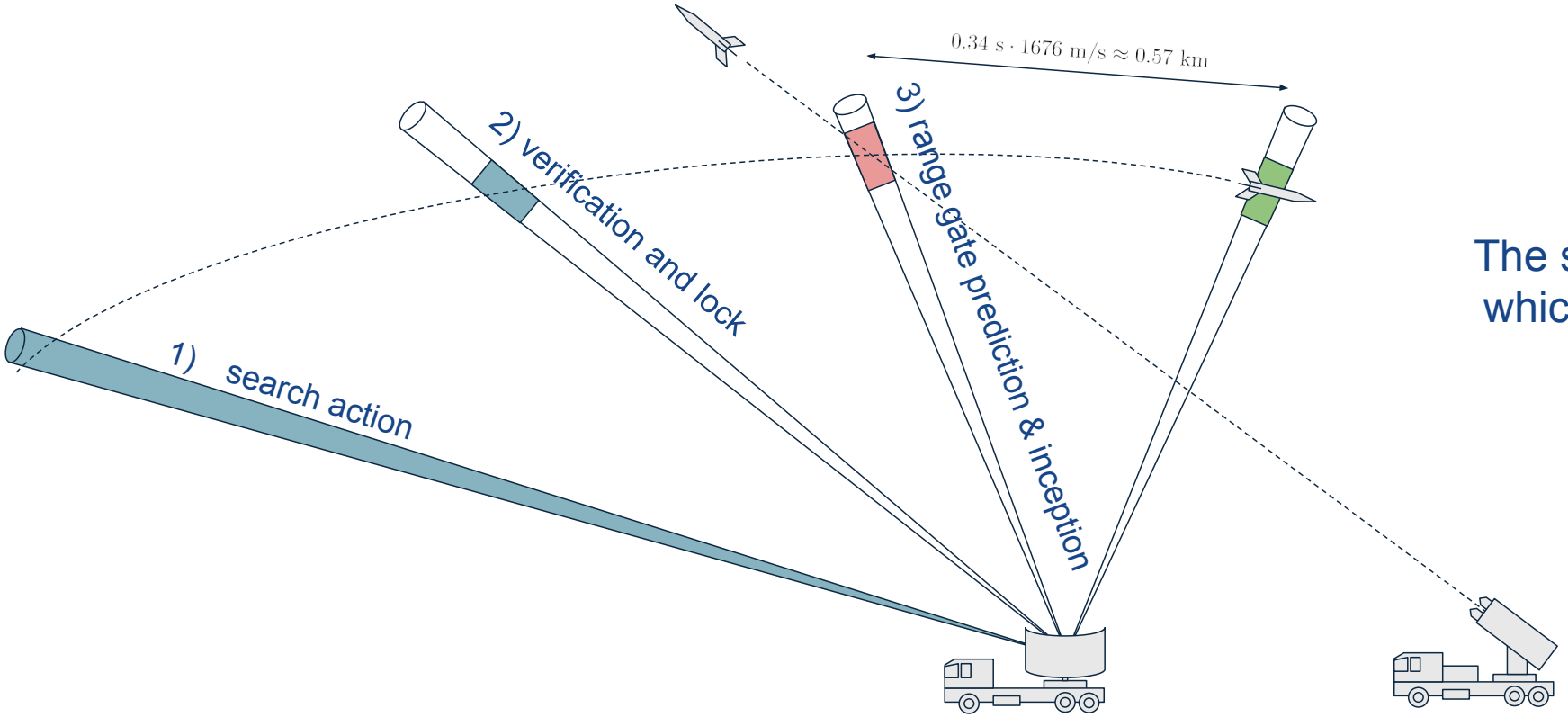
Dharan, Saudi Arabia, on February 25, 1991
GAO/IMTEC-92-26

Δt =**1/10** s in **24 bit** fixed point arithmetics.



The Patriot Missile Failure

Dharan, Saudi Arabia, on February 25, 1991
GAO/IMTEC-92-26



Δt = **1/10** s in **24 bit** fixed point arithmetics.

$0.1_{10} = 0.0001100110011001100110011001100\ldots_2$

$0.1_{10} \approx 0.000110011001100110011001_2 (011001100\ldots)_2$

$\approx 0.000000095_{10}$

$\approx$ 95 ns

The system was running for **100h** which lead to cumulative error of **0.34s.**

Multiple casualties

All because of poor handling of rounding error

Motivation

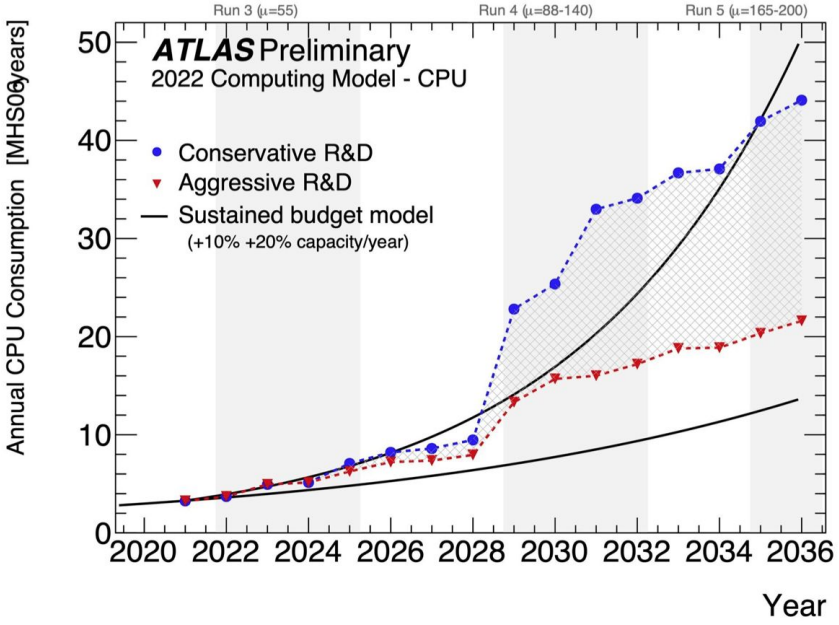
LHC is going to be upgraded to High Luminosity LHC by **2030.**

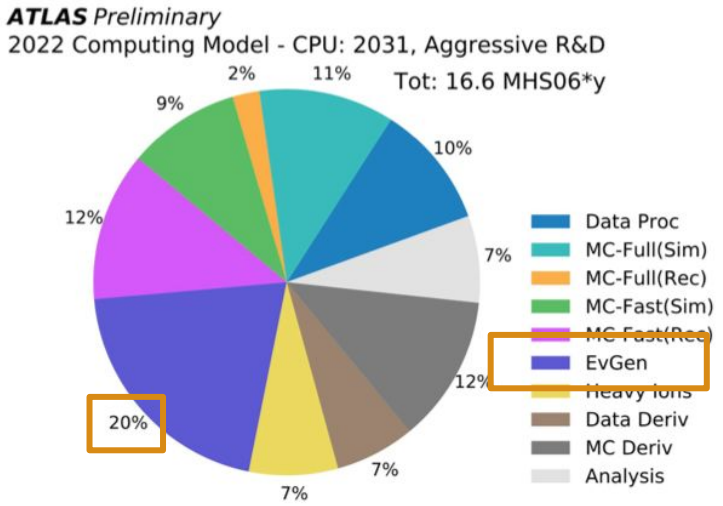
HL-LHC will increase the number of particle collisions **5x-7.5x.**

This will raise **significantly** the **computing** demands.

Will need unprecedented volumes of **event generation.**

Good that we have free lunch, right?
(Denard, Moore scaling)



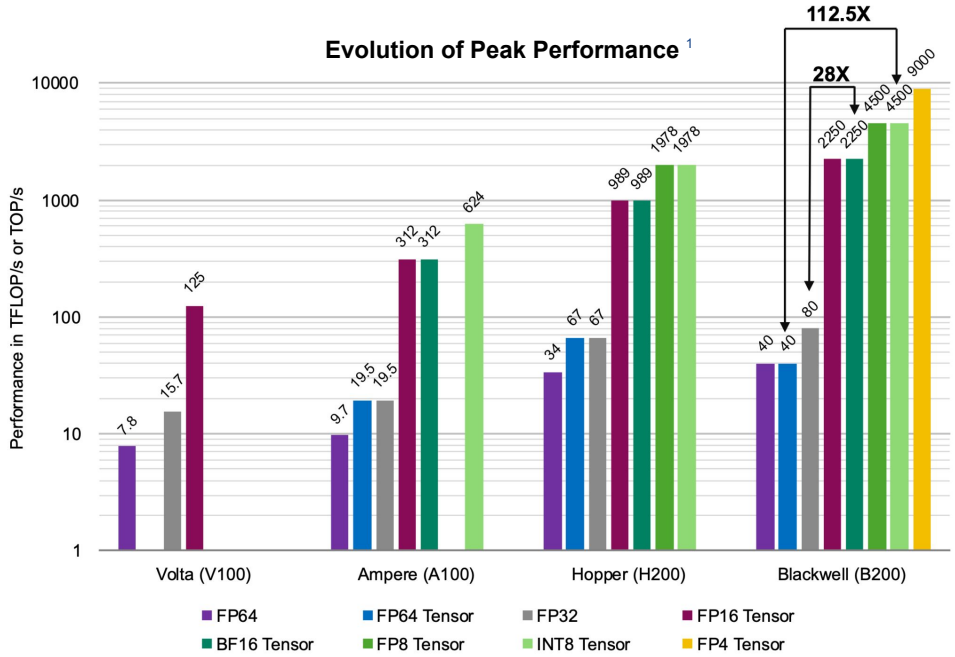


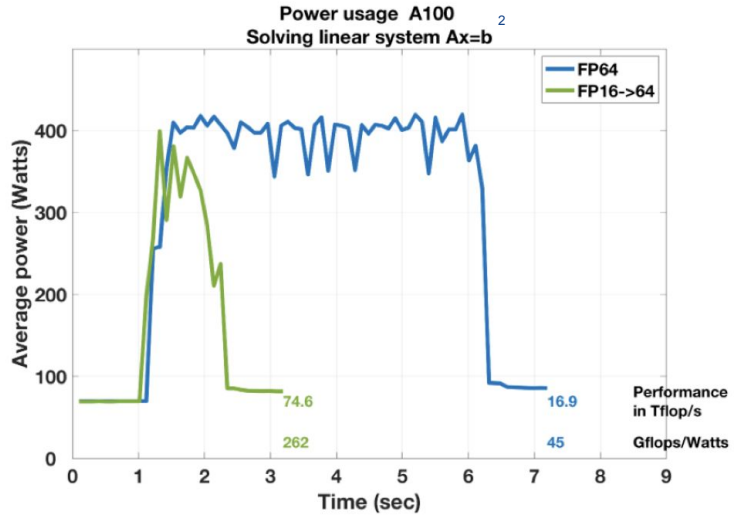
[1] ATLAS Software and Computing HL-LHC Roadmap - [URL]

Motivation

- Increasing floating-point precision (more bits) means higher **energy consumption and computational cost**.
  - $E(FP64)/E(FP32) \approx 5$
- **Quadruple precision** is not natively supported on most modern hardware.
- **GPU market** is increasingly driven by **AI applications**, favouring *lower precision* (FP16, BFLOAT16,...) to maximise throughput.

→ **We must rethink our approach to precision and performance trade-offs.**





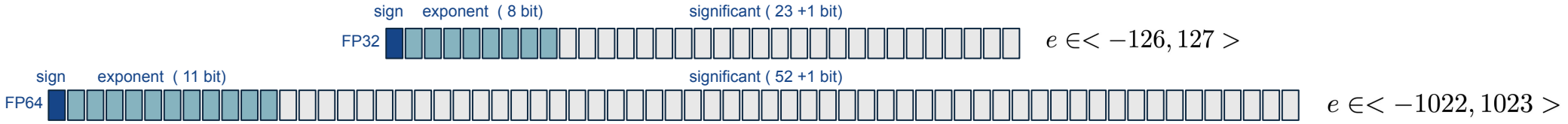
[1] Floating Point Emulation in NVIDIA Math Libraries - Rodriguez S. - OFPP workshop 2025, CERN - [URL]
[2] Hardware Trends Impacting Floating-Point Computations In Scientific Applications - Dongarra J. - math.NA, 2024 [URL]

Supplement - IEEE-754

$$x = (-1)^{sgn} \sum_{i=0}^{p-1} d_i b^{-i} b^e$$

$$sgn \in \{0, 1\}$$
$$d_i \in \{0, ..., b-1\}$$
$$e_{min} \leq e \leq e_{max}$$
$$e_{min} = -(e_{max} - 1)$$

$$F^*(b, p, e_{min}, e_{max})$$



Rounding

- RN, RZ, RU, RD
- correct rounding for +,-,*,/, sqrt()
- GRS system

for RN $\longrightarrow$ $|\mathrm{fl}(x) - x| \leq 0.5 \cdot \mathrm{ulp}(x)$

Supplement

Floating point arithmetics (operations) is **commutative** but not **associative** nor **distributive**.

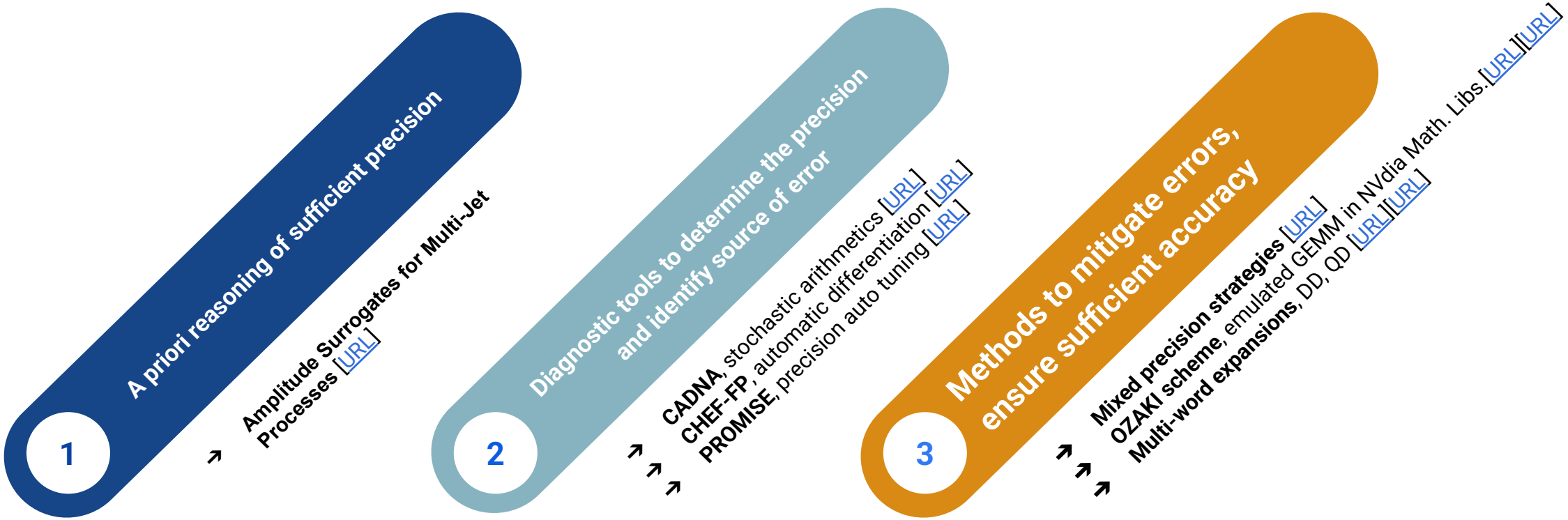
Not every number can be represented.

Addition of two numbers of varying magnitude -> loss of **low** order bits.

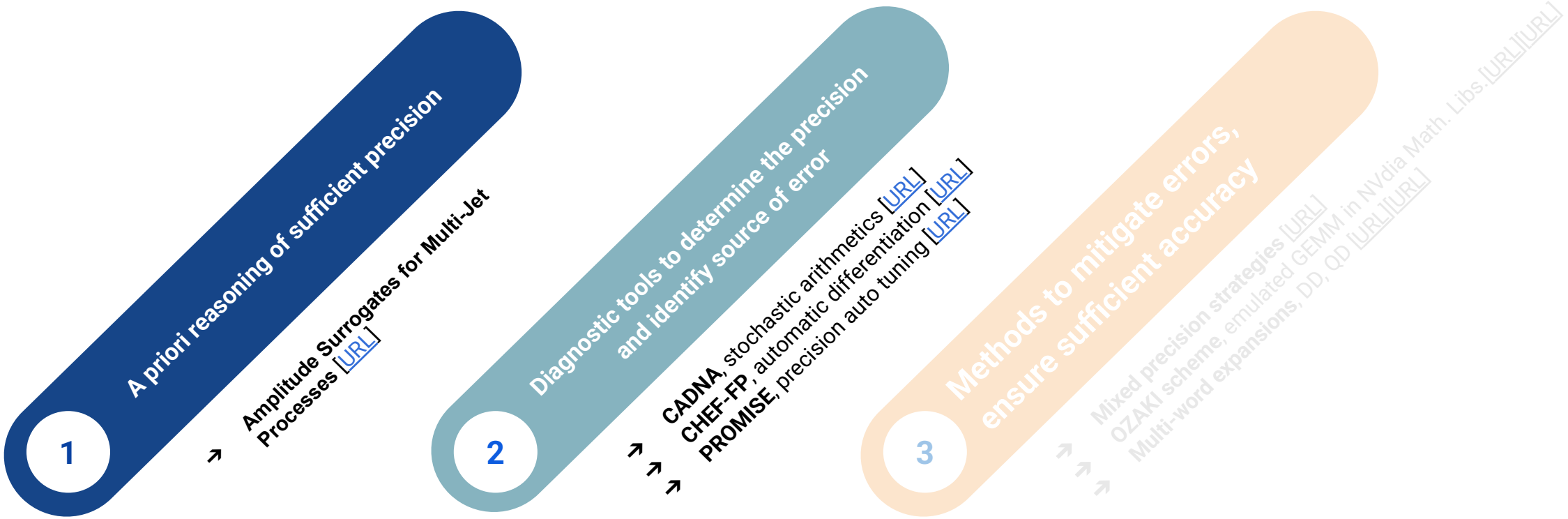
Subtracting two numbers of similar size -> loss of **higher** bits (up to catastrophic cancellation).

Loss of **significant** digits is independent of floating point **precision.**

Need to know WHAT, WHERE and HOW?



Need to know WHAT, WHERE and HOW?



What accuracy do we want?

Numerical precision at LO uncertainties dominate:

- Scale variation: **~50%** (norm), **≥10%** (shapes)
- PDFs: **~3%**, larger in kinematic tails

Precision benchmark

- Gaussian **3σ** (99.7%) **conservative < 10^{-3}** of events may have lower accuracy **< 99.9%**

Error propagation limits

- **Optimistic:** uncorrelated errors → Monte Carlo cancellation
- **Pessimistic:** fully correlated bias → **1% systematic** on cross section

Goal is 3 significant digits for > 99.9% of events

[1] Amplitude Surrogates for Multi-Jet Processes [URL]

Determine the accuracy of output

- **Static analysis**
  - All possible input values taken into account, no execution
  - **FLDLib**

- **Interval arithmetics**
  - Guarantees bounds of the computed result
  - Usually overestimation, not suitable for large codes
  - **INTLAB**

- **Probabilistic approach**
  - No algorithmic modification needed
  - Estimation for any variable
  - **CADNA**

3.14918

Discrete stochastic arithmetics (DSA)

- Each floating point **operation** is computed **n times.**
- For each operation **random rounding mode** is set.
- Commonly to obtain the number of **correct digits, Student test** with **confidence level 95%** is performed on the n samples.
- **Computational zero @0** (0 correct digits)

$$A \oplus B \to R$$

$$A_1 \oplus B_1 \to R_1$$
$$A_2 \oplus B_2 \to R_2$$
$$A_3 \oplus B_3 \to R_3$$
$$\dots$$

$\otimes$ floating point multiplication	$\otimes_{+\infty}$ rounding towards +inf
$\oplus$ floating point addition	$\otimes_{-\infty}$ rounding towards -inf

CADNA [URL]

- DSA for **C++** and **Fortran** codes
- **Stochastic type** composed of **3 floating points and 1 int**
- **half, float, double (+ arbitrary)**
- All math **operations** are **overloaded**
- Support for **MPI, OpenMP** and **GPU**

$$a \oplus_{+\infty} b = -(-a \oplus_{-\infty} -b)$$
$$a \otimes_{+\infty} b = -(a \otimes_{-\infty} -b)$$

One **CADNA** execution - **accuracy** of any result and list of **instabilities**

$\otimes$ floating point multiplication	$\otimes_{+\infty}$ rounding towards +inf
$\oplus$ floating point addition	$\otimes_{-\infty}$ rounding towards -inf

Precision auto-tuning

- **Static**
  - e.g. Taylor expansions
  - Not suitable to large codes
  - **FPTaylor**

- **Dynamic**
  - comparison with highest precision type result
  - **ADAPT** (algo. diff.)**, PROMISE** (DSA)

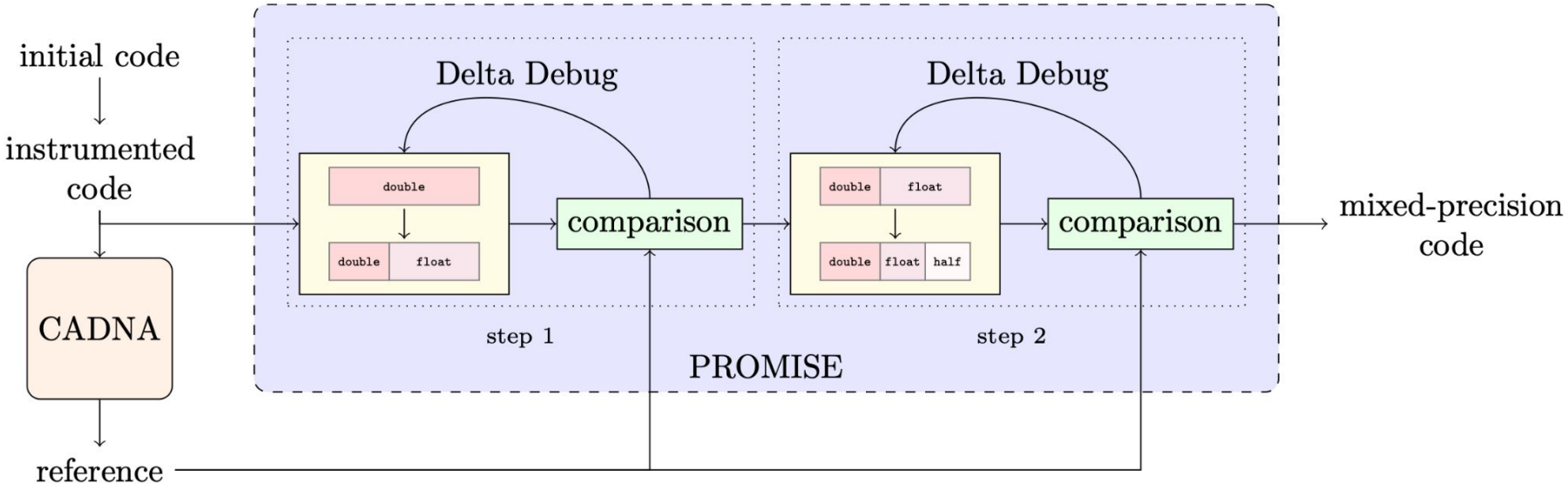
~~float A;~~

double A;

PROMISE[URL]

Uses CADNA for check of the highest type result

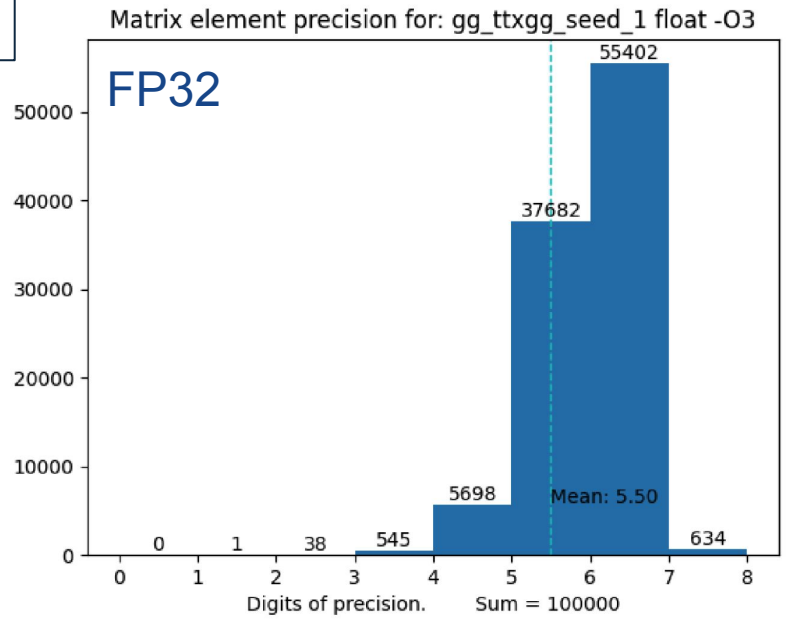
Built on DELTA DEBUG algorithm O(nlog(n)) for n vars

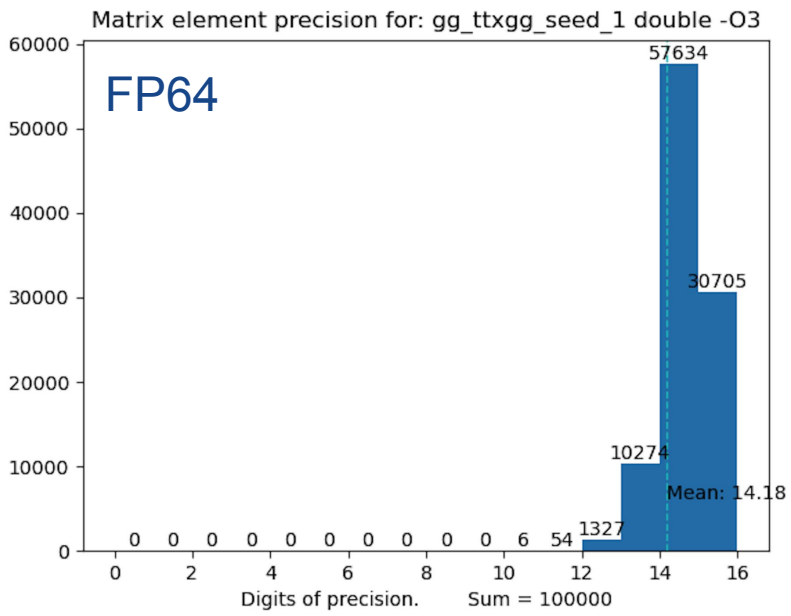


CADNA in MadGraph5

We assume that momenta enter the ME calculation with infinite precision and use just the "standalone" version.

g g → t t~ g g

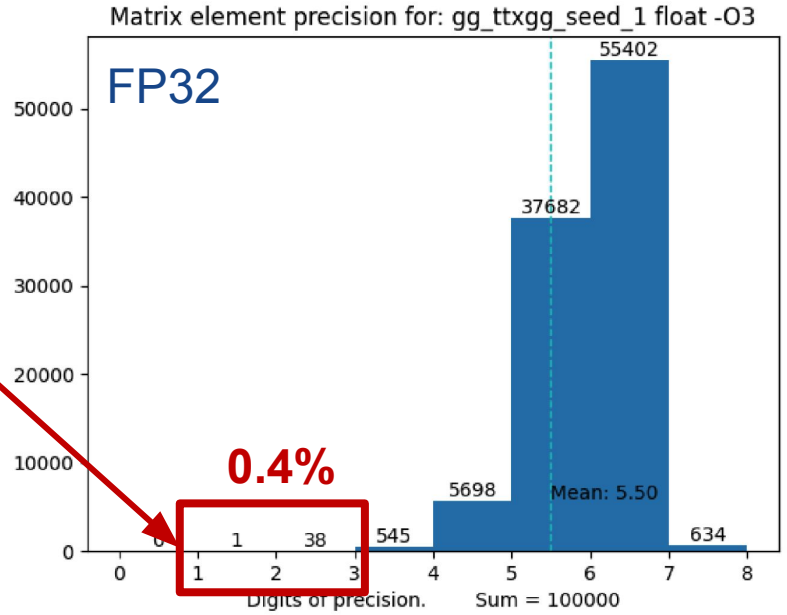




CADNA in MadGraph5

g g → t t~ g g

Only those are the problematic ones, what are the properties?

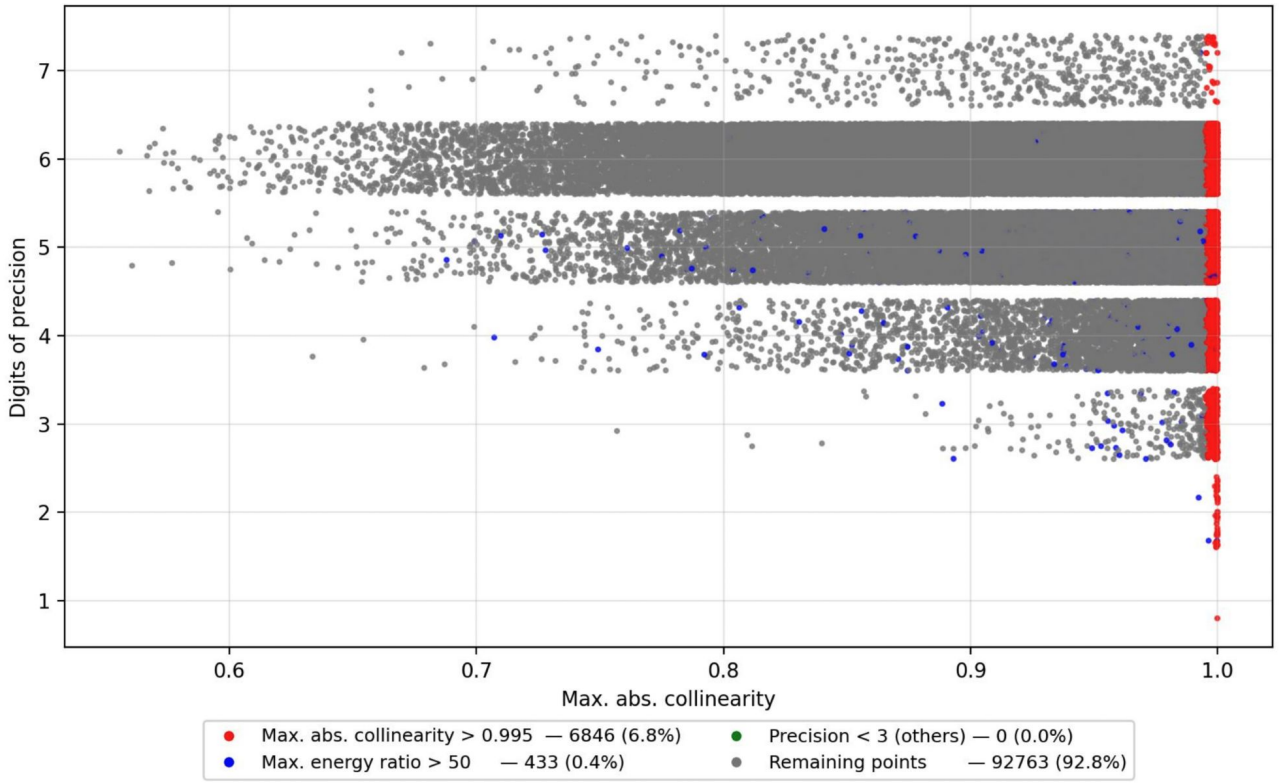


They might trip the phase-space integrator - because of those we need to run in FP64 everything?

Momenta study

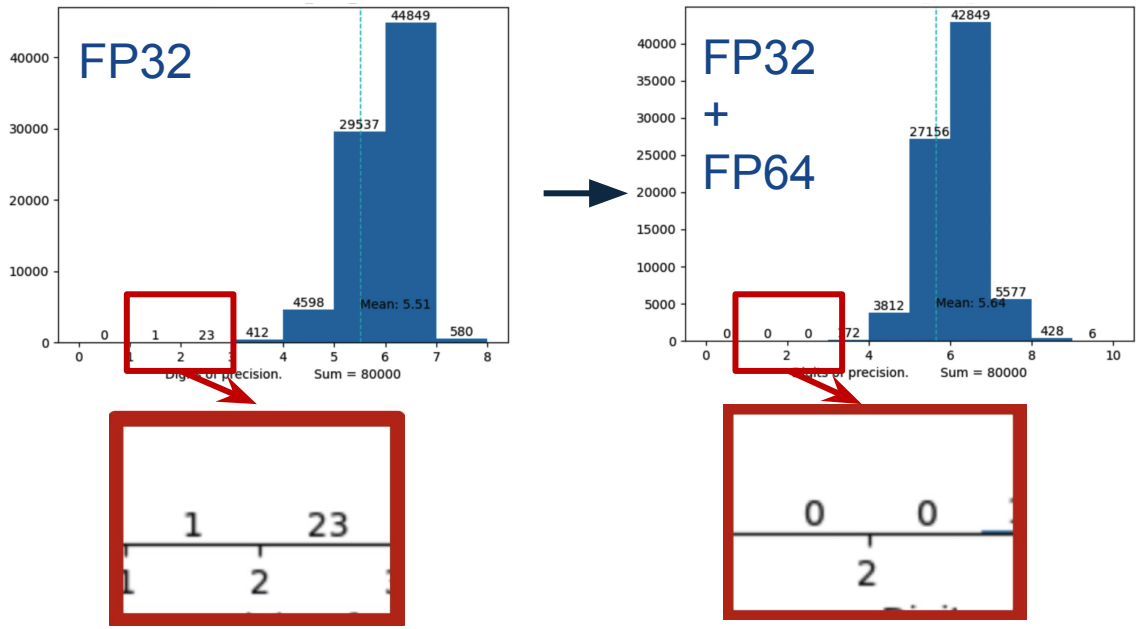
What do the bad momenta have in common?

$$\text{soft} = \frac{\max_{i \in \hat{N}} E_i}{\min_{j \in \hat{N}} E_j}, \quad \text{coll} = \max_{i<j} \frac{|\vec{p}_i \cdot \vec{p}_j|}{\|\vec{p}_i\| \, \|\vec{p}_j\|}$$



Separation of the momenta by soft.+coll.

g g → t t~ g g

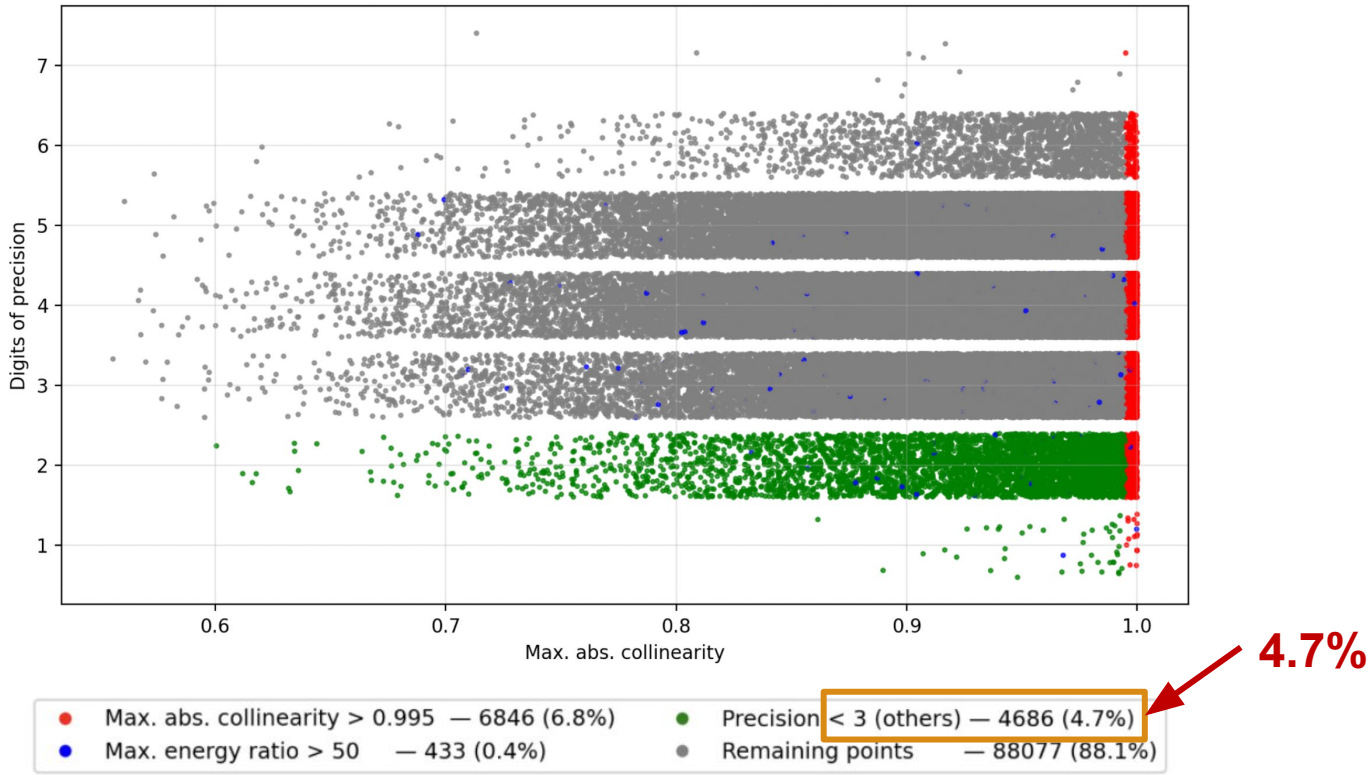


0.4% **0.0%**

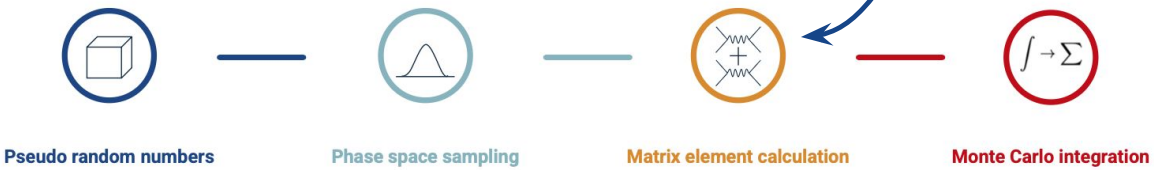
Momenta study

Not all the processes have the same behaviour

u d~ → w+ w- c s~ QCD=0



Where do we lose the accuracy?



Bottleneck - ported to C++ and offloaded on GPU/vectorized

Thanks to Daniele Massaro

Where do we lose the accuracy?

Helicity sum

Color sum

Where do we lose the accuracy?

Helicity sum

$$1 - \cos\theta \xrightarrow{\theta \to 0} \text{numerical instability}$$

$$\frac{1}{p_\mu p^\mu - M^2} \xrightarrow{p^2 \to M^2} \text{numerical instability}$$

+ Gauge dependence

Color sum

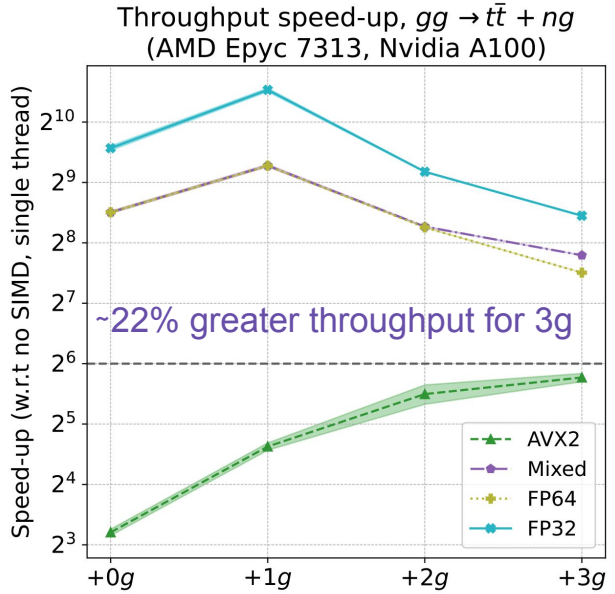
Where do we lose the accuracy?

Helicity sum | Color sum

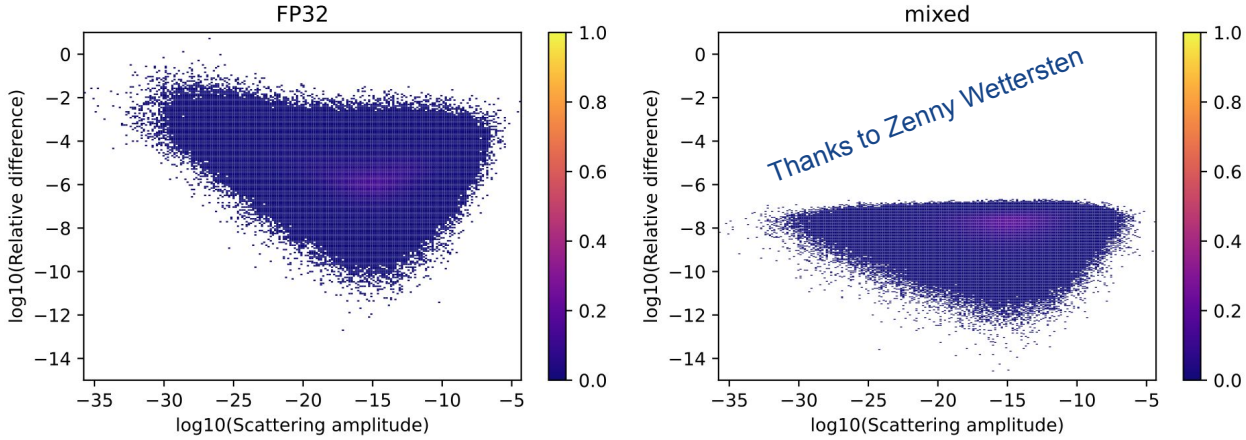
$$1 - \cos\theta \xrightarrow{\theta \to 0} \text{numerical instability}$$

$$\frac{1}{p_\mu p^\mu - M^2} \xrightarrow{p^2 \to M^2} \text{numerical instability}$$

+ Gauge dependence



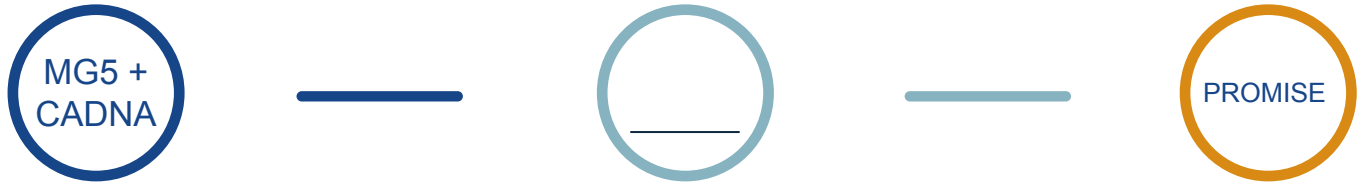
$uu \to H + 4j$



[1] Data-parallel leading-order event generation in MadGraph5_aMC@ NLO [PDF]

PROMISE in MadGraph5

Some processes violate previous assumption of possibility of separation of the momenta.



Find “bad” momenta

CADNA instrumented MG5 to generate the violating (<3s.d.) momenta for further analysis.

Boilerplate for only the ME

Automatic creation of a boilerplate to run just the ME calculation for easier and faster PROMISE processing.

Each function is templated so a type can be determined.

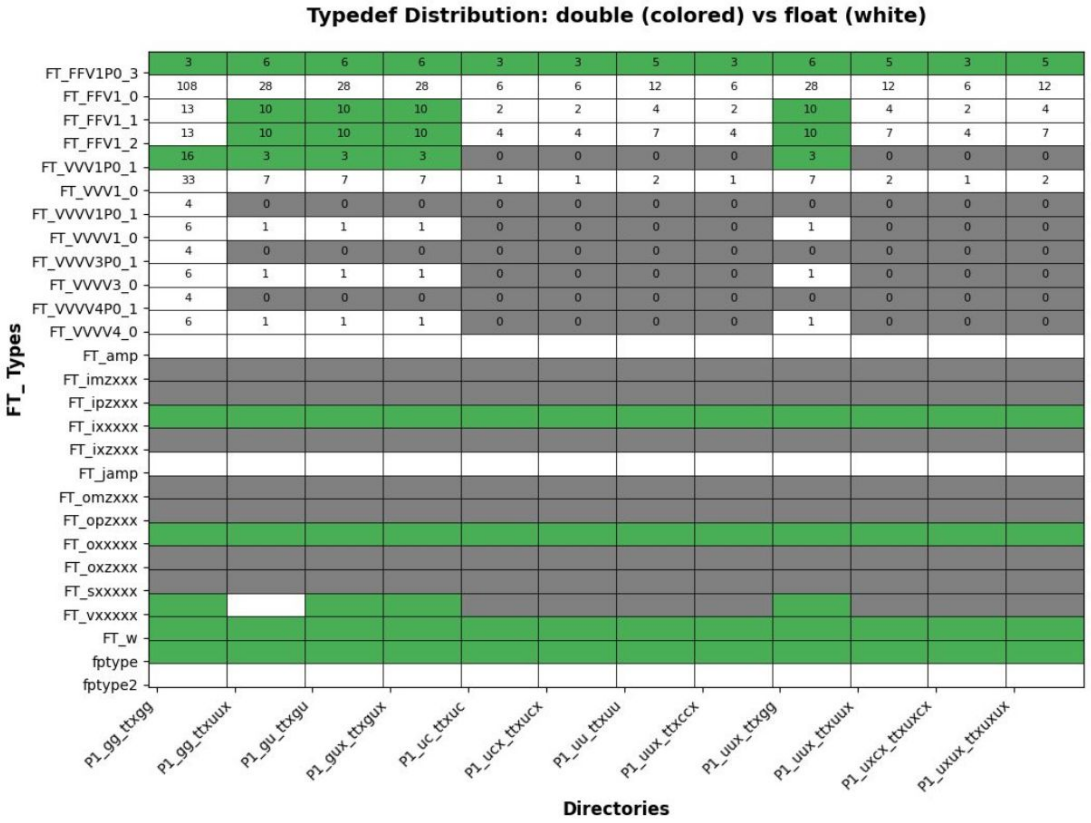
PROMISE analysis

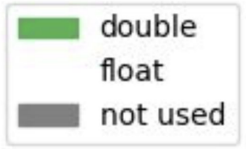
Lastly, PROMISE is employed to automatically determine the sufficient type to ensure the 3 sig. dig. of a result.

PROMISE in MadGraph5

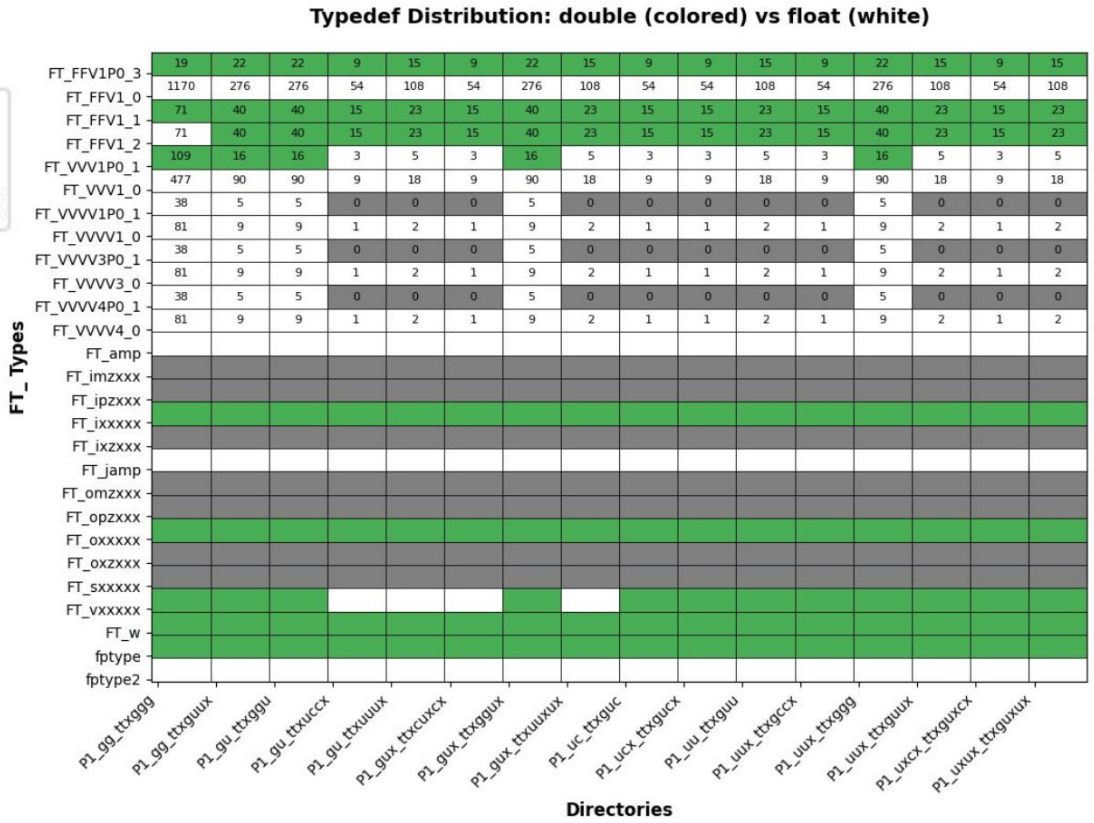
Example of FFV

p p → t t~ + 2j





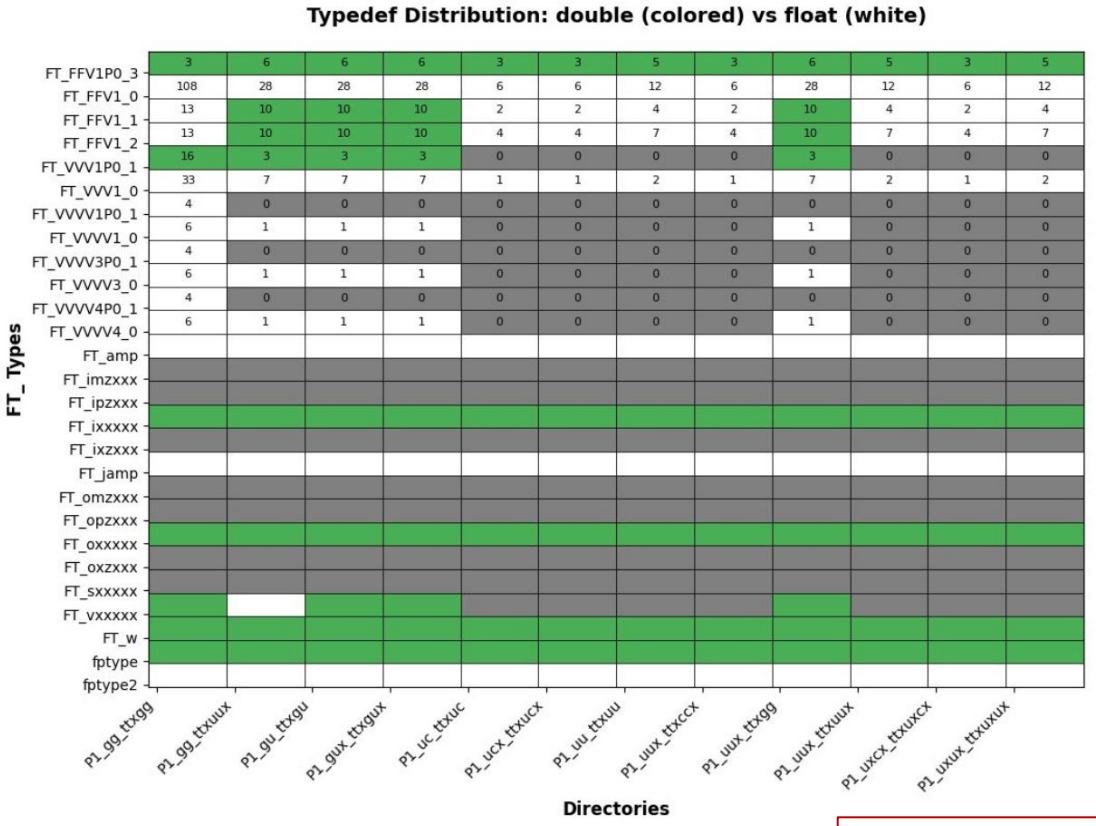
p p → t t~ +3j

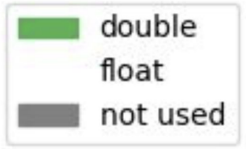


PROMISE in MadGraph5

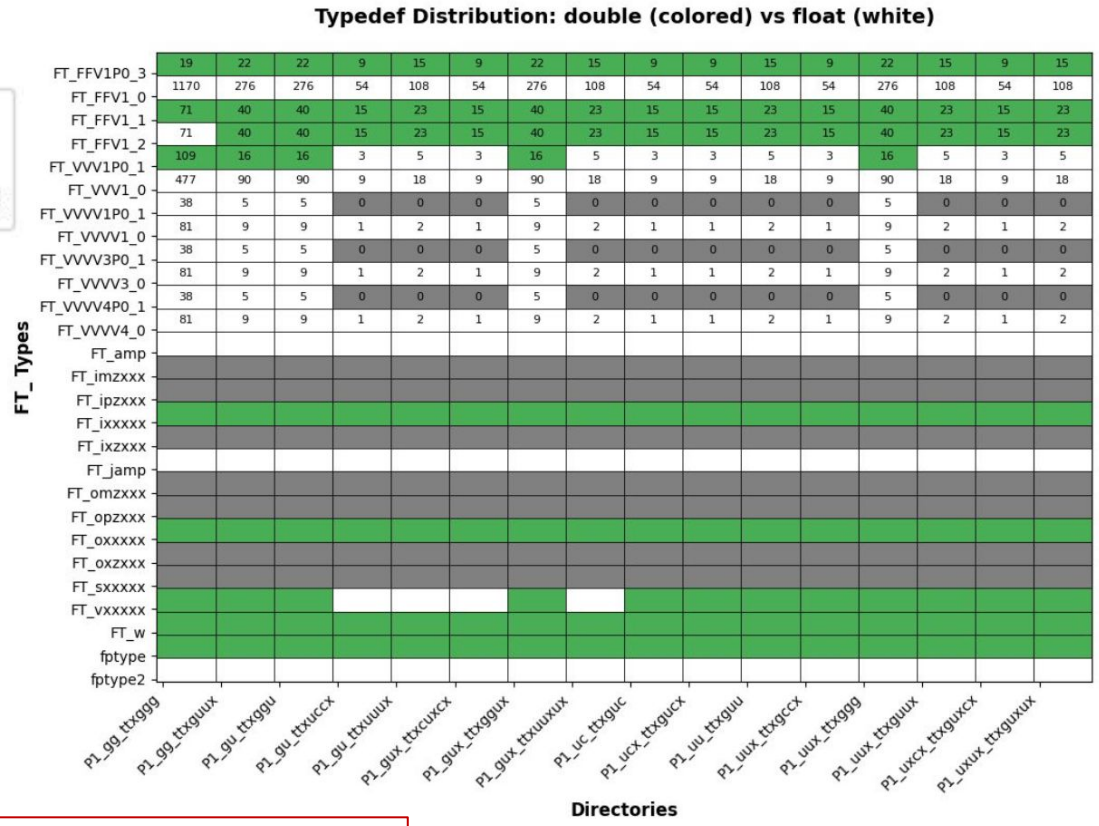
Example of FFV

p p → t t~ + 2j





p p → t t~ +3j



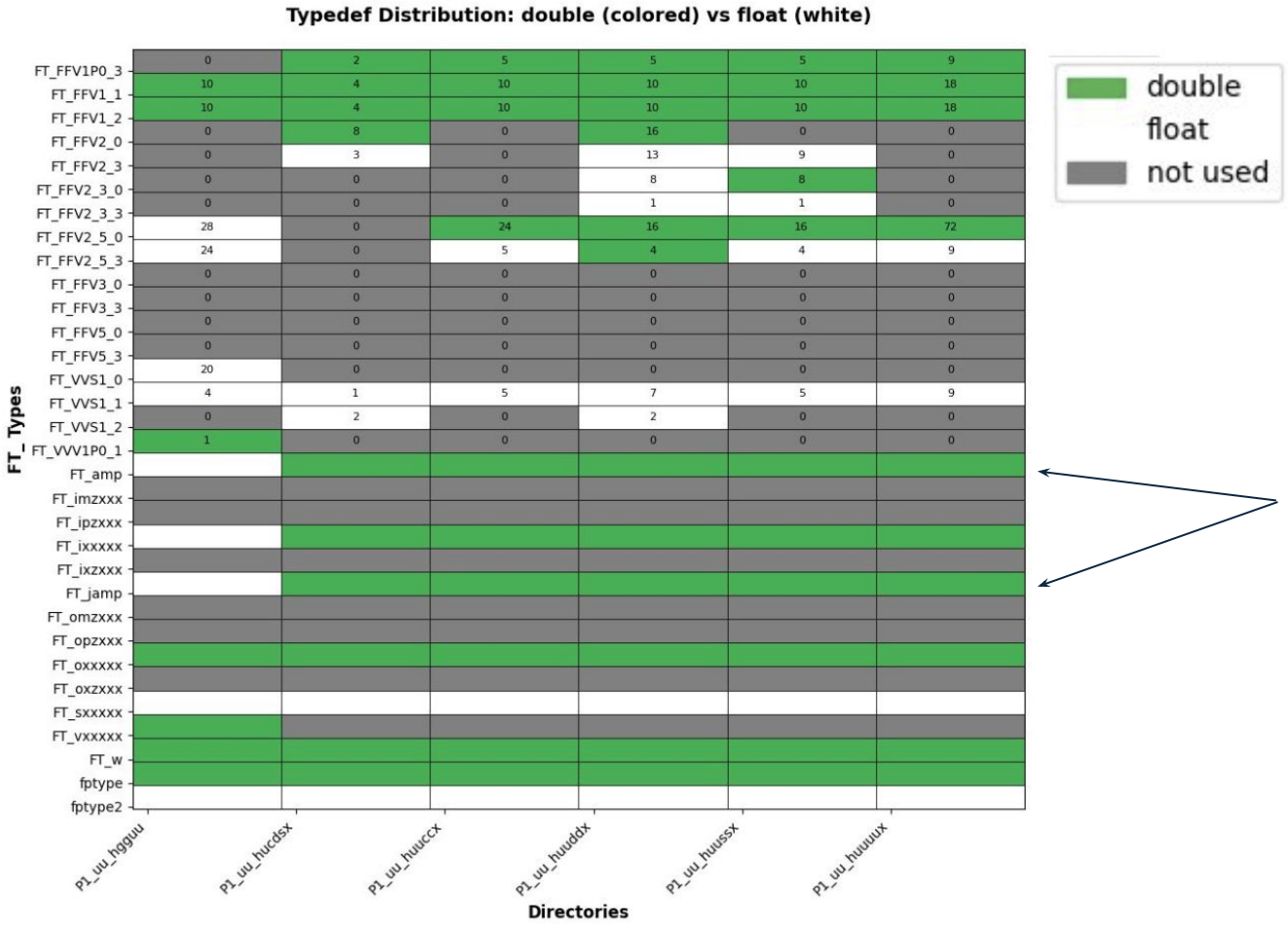
FFV1_1, FFV1_2, FFV1P0_3, VVV1P0_1

Similar for lower mult. (jets)

PROMISE in MadGraph5

Example of FFV

$uu \rightarrow H + 4j$

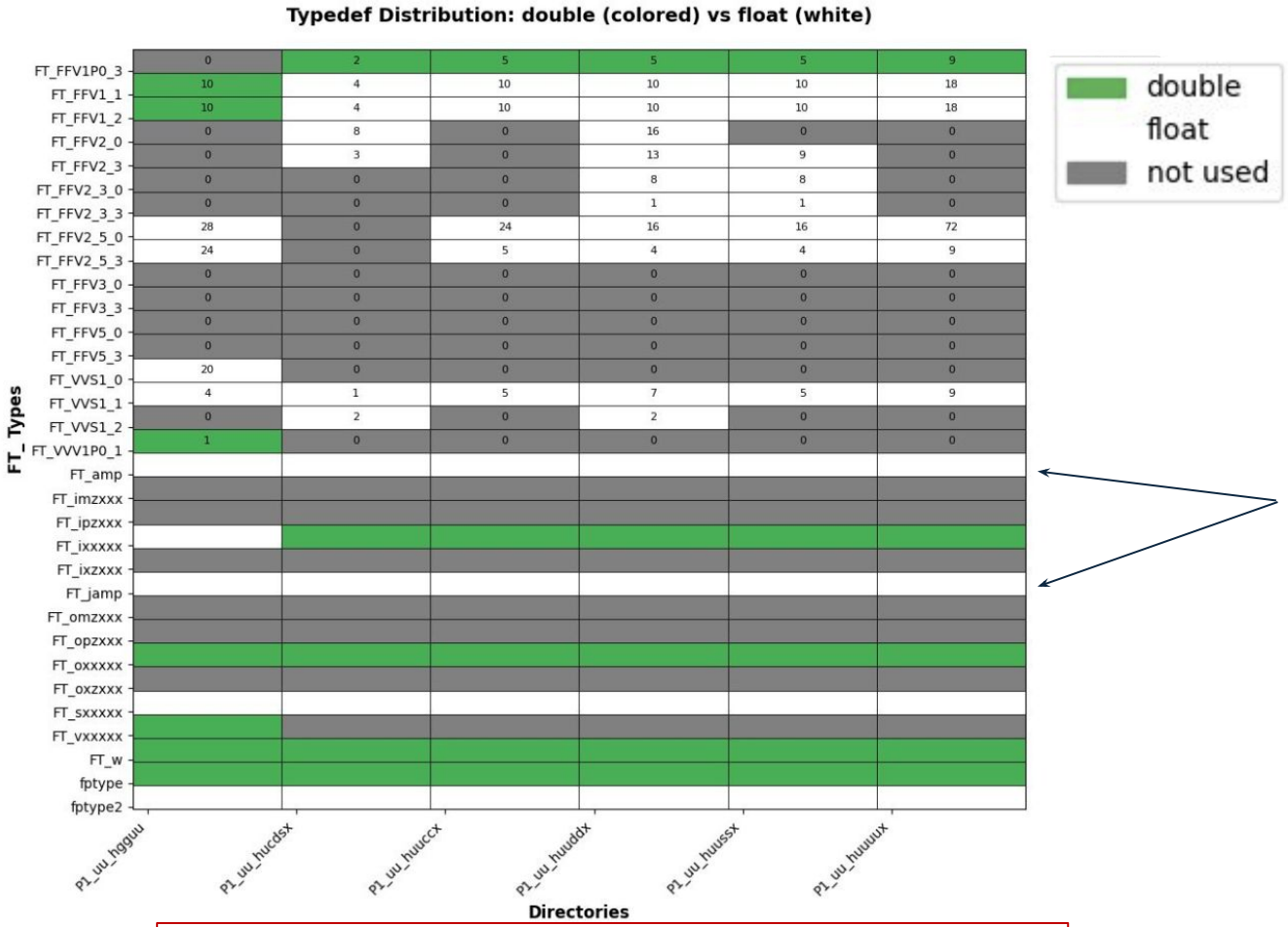


Summation between diagrams

Example of FFV

PROMISE in MadGraph5

uu → H + 4j
axial gauge



Summation between diagrams in FP32

FFV1_1, FFV1_2, FFV1P0_3, VVV1P0_1

Closer look into the vertex functions

Example of FFV

```
template<class W_ACCESS, class C_ACCESS>
__device__ void
FFV1P0_3( const fptype allF1[],
          const fptype allF2[],
          const fptype allCOUP[],
          const double Ccoeff,
          const fptype M3,
          const fptype W3,
          fptype allV3[] )
{
 const cxtype_sv* F1 = W_ACCESS::kernelAccessConst( allF1 );
 const cxtype_sv* F2 = W_ACCESS::kernelAccessConst( allF2 );
 const cxtype_sv COUP = C_ACCESS::kernelAccessConst( allCOUP );
 cxtype_sv* V3 = W_ACCESS::kernelAccess( allV3 );
 const cxtype cI = cxmake( 0., 1. );
 V3[0] = +F1[0] + F2[0];
 V3[1] = +F1[1] + F2[1];
 const fptype_sv P3[4] = { -cxreal( V3[0] ), -cxreal( V3[1] ), -cximag( V3[1] ), -cximag( V3[0] ) };
 const cxtype_sv denom = Ccoeff * COUP / ( ( P3[0] * P3[0] ) - ( P3[1] * P3[1] ) - ( P3[2] * P3[2] ) - ( P3[3] * P3[3] ) - M3 * ( M3 - cI * W3 ) );
 V3[2] = denom * ( -cI ) * ( F1[2] * F2[4] + F1[3] * F2[5] + F1[4] * F2[2] + F1[5] * F2[3] );
 V3[3] = denom * ( -cI ) * ( -F1[2] * F2[5] - F1[3] * F2[4] + F1[4] * F2[3] + F1[5] * F2[2] );
 V3[4] = denom * ( -cI ) * ( -cI * ( F1[2] * F2[5] + F1[5] * F2[2] ) + cI * ( F1[3] * F2[4] + F1[4] * F2[3] ) );
 V3[5] = denom * ( -cI ) * ( -F1[2] * F2[4] - F1[5] * F2[3] + F1[3] * F2[5] + F1[4] * F2[2] );
 return;
}
```

* After line by line PROMISE analysis

Example of FFV

Closer look into the vertex functions

```
template<class W_ACCESS, class C_ACCESS>
__device__ void
FFV1P0_3( const fptype allF1[],
          const fptype allF2[],
          const fptype allCOUP[],
          const double Ccoeff,
          const fptype M3,
          const fptype W3,
          fptype allV3[] )
{
 const cxtype_sv* F1 = W_ACCESS::kernelAccessConst( allF1 );
 const cxtype_sv* F2 = W_ACCESS::kernelAccessConst( allF2 );
 const cxtype_sv COUP = C_ACCESS::kernelAccessConst( allCOUP );
 cxtype_sv* V3 = W_ACCESS::kernelAccess( allV3 );
 const cxtype cI = cxmake( 0., 1. );
 V3[0] = +F1[0] + F2[0];
 V3[1] = +F1[1] + F2[1];
 const fptype_sv P3[4] = { -cxreal( V3[0] ), -cxreal( V3[1] ), -cximag( V3[1] ), -cximag( V3[0] ) };
 const cxtype_sv denom = Ccoeff * COUP / ( ( P3[0] * P3[0] ) - ( P3[1] * P3[1] ) - ( P3[2] * P3[2] ) - ( P3[3] * P3[3] ) - M3 * ( M3 - cI * W3 ) );
 V3[2] = denom * ( -cI ) * ( F1[2] * F2[4] + F1[3] * F2[5] + F1[4] * F2[2] + F1[5] * F2[3] );
 V3[3] = denom * ( -cI ) * ( -F1[2] * F2[5] - F1[3] * F2[4] + F1[4] * F2[3] + F1[5] * F2[2] );
 V3[4] = denom * ( -cI ) * ( -cI * ( F1[2] * F2[5] + F1[5] * F2[2] ) + cI * ( F1[3] * F2[4] + F1[4] * F2[3] ) );
 V3[5] = denom * ( -cI ) * ( -F1[2] * F2[4] - F1[5] * F2[3] + F1[3] * F2[5] + F1[4] * F2[2] );
 return;
}
```

FP64*

Same behaviour for VVV1P0_1

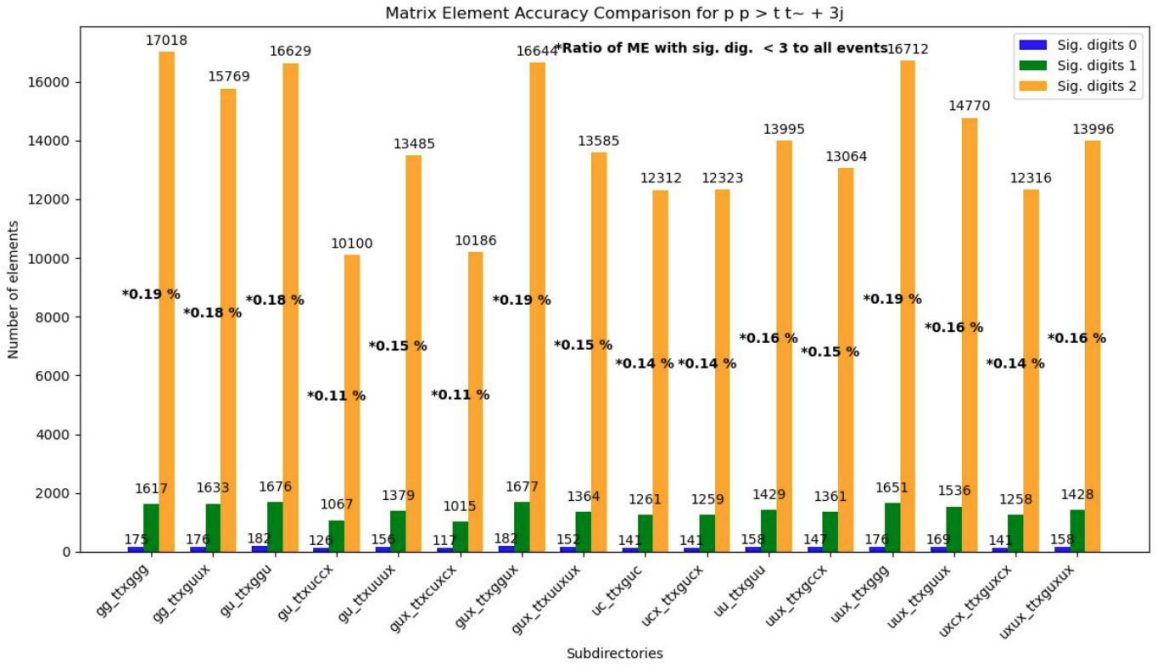
$$\frac{1}{p_\mu p^\mu - M^2} \xrightarrow{p^2 \to M^2} \text{numerical instability}$$

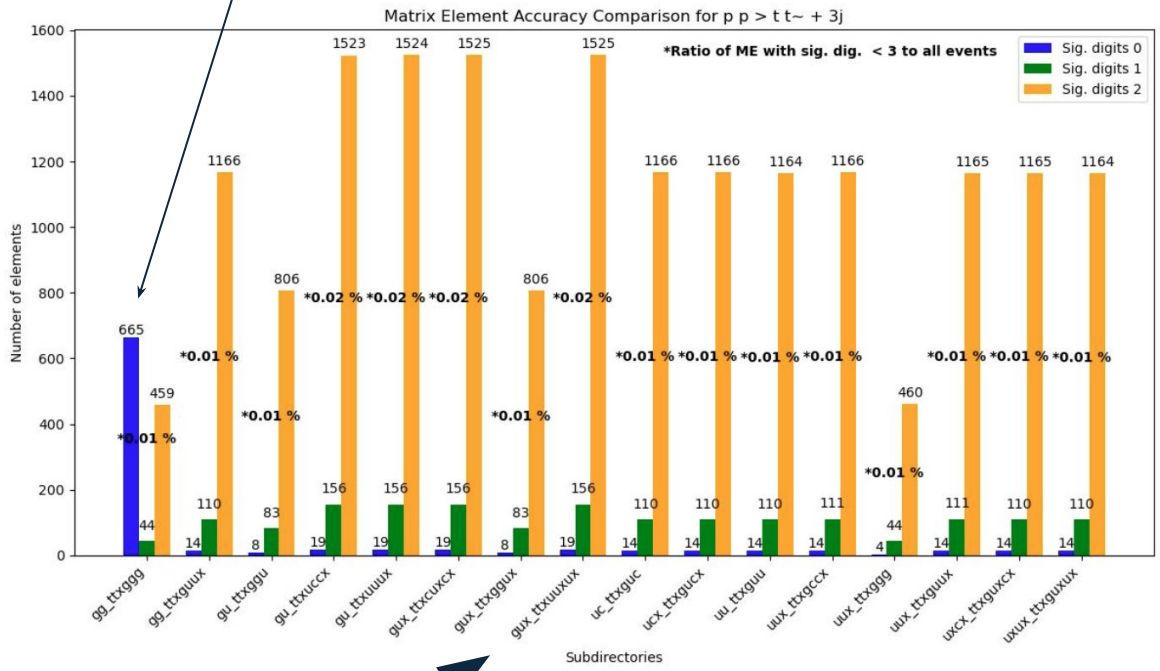
* After line by line PROMISE analysis

What if we set denom to 1.0 ?

p p → t t~ +3j

To be investigated. (Overflow not in normal process)



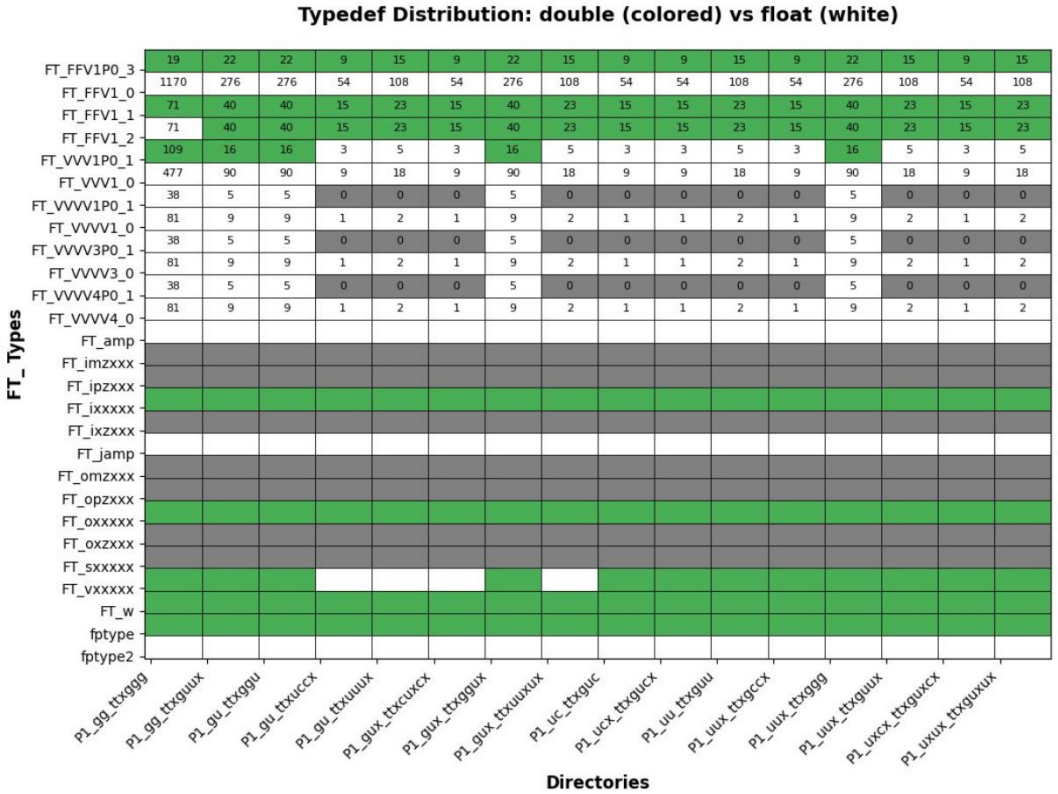


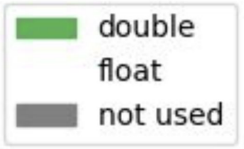
From <0.2% to <0.02%

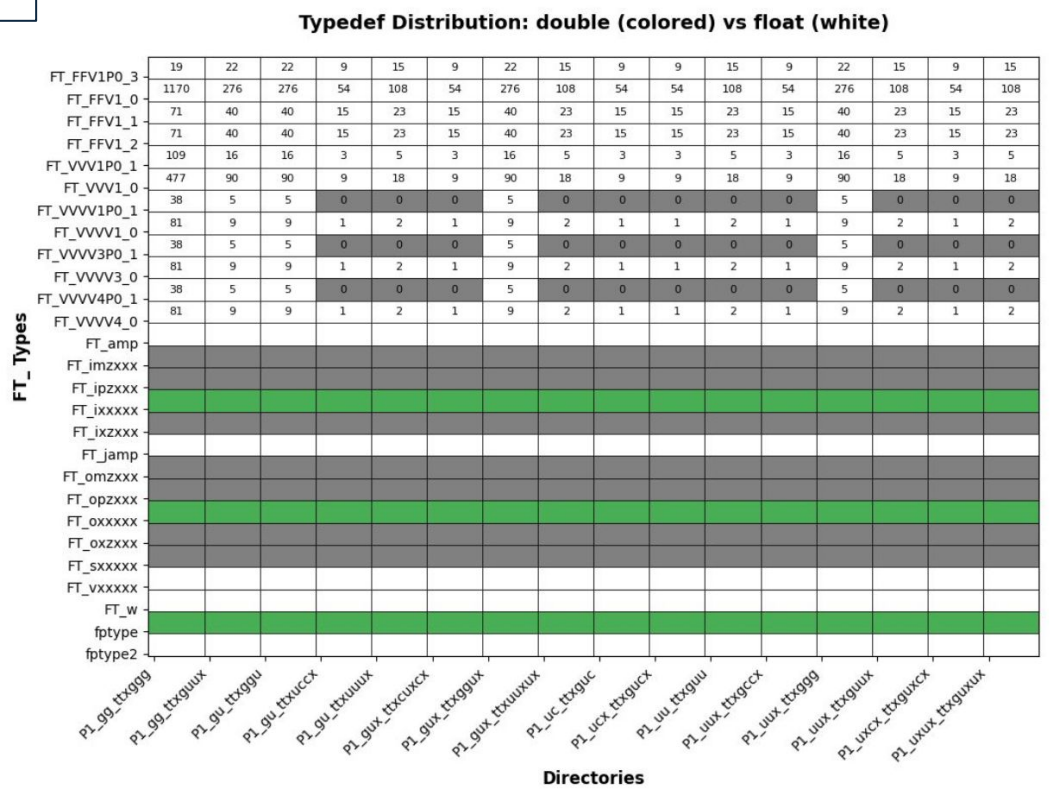
What if we set denom to 1.0 ?

Example of FFV

p p → t t~ +3j







Only the incoming* in FP64

Separation?

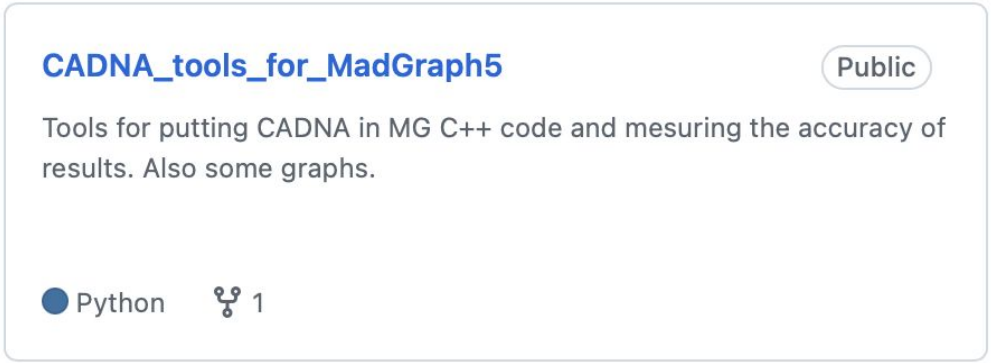
*We need to optimize them next.

Future steps

Perform a line-by-line analysis with PROMISE of rest of functions flagged for double-precision.

Apply pattern-based algorithmic remedies (e.g., compensated methods, arithmetic expansions) adapted to complex arithmetic.

FD gauge (when available in C++ version). Phase space sampling examination. …



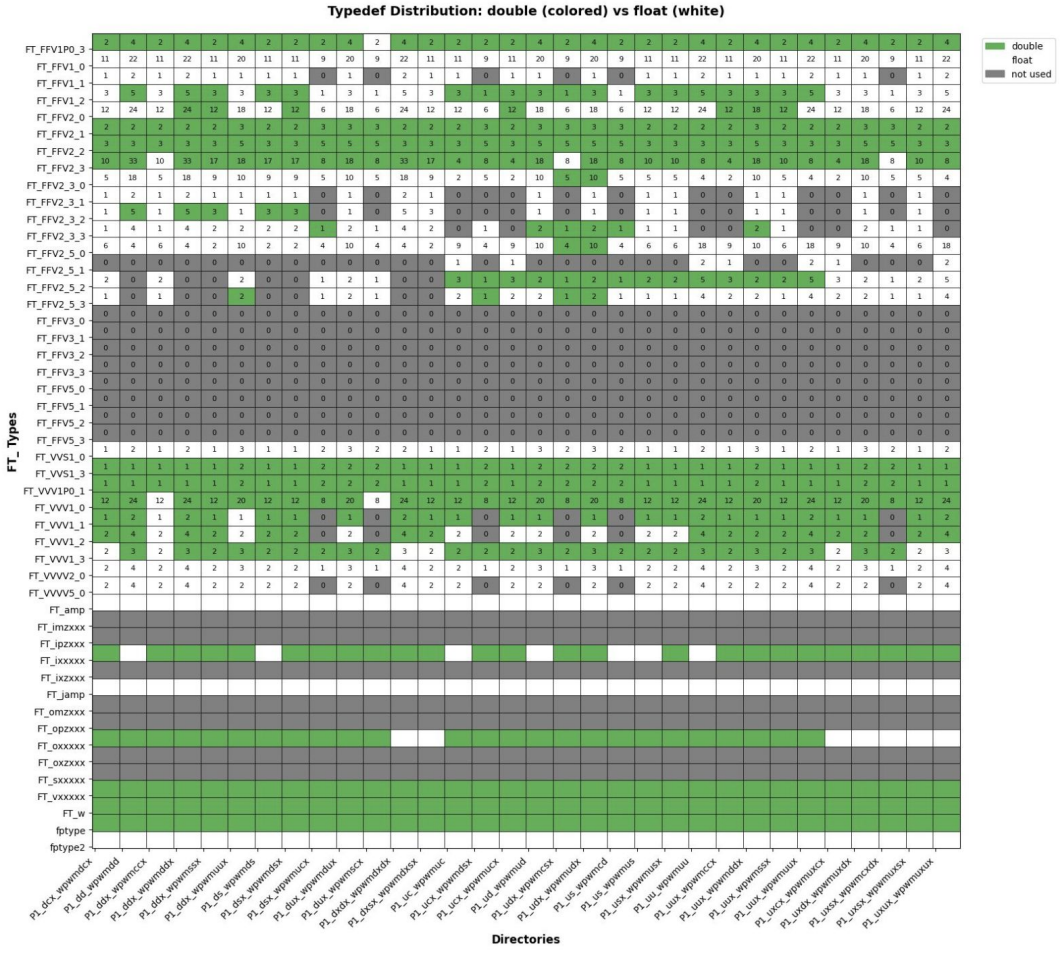
This work has been funded by the Eric & Wendy Schmidt Fund for Strategic Innovation through the CERN Next Generation Triggers project under grant agreement number SIF-2023-004.

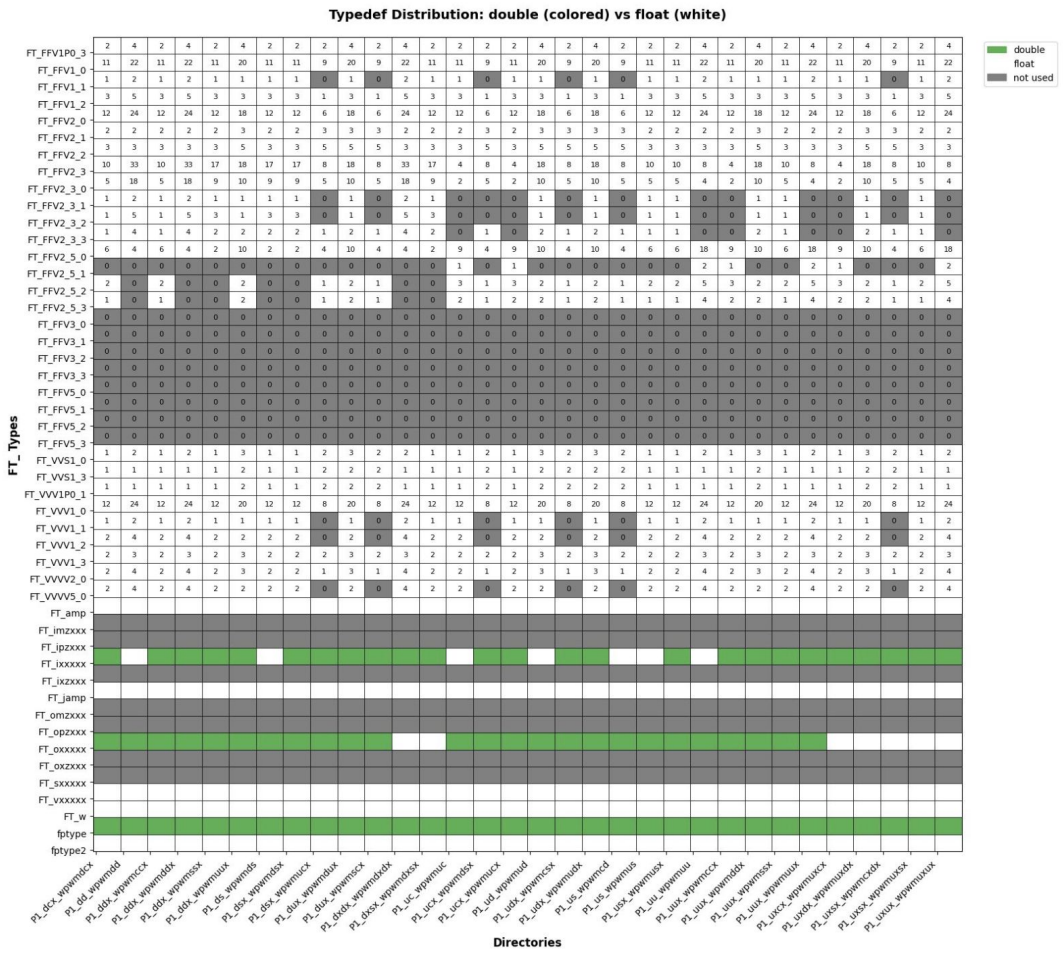

CERN
NexTGen
Next Generation Triggers

Backup

p p → W+ W- +2j QCD=0j

denom = 1.0

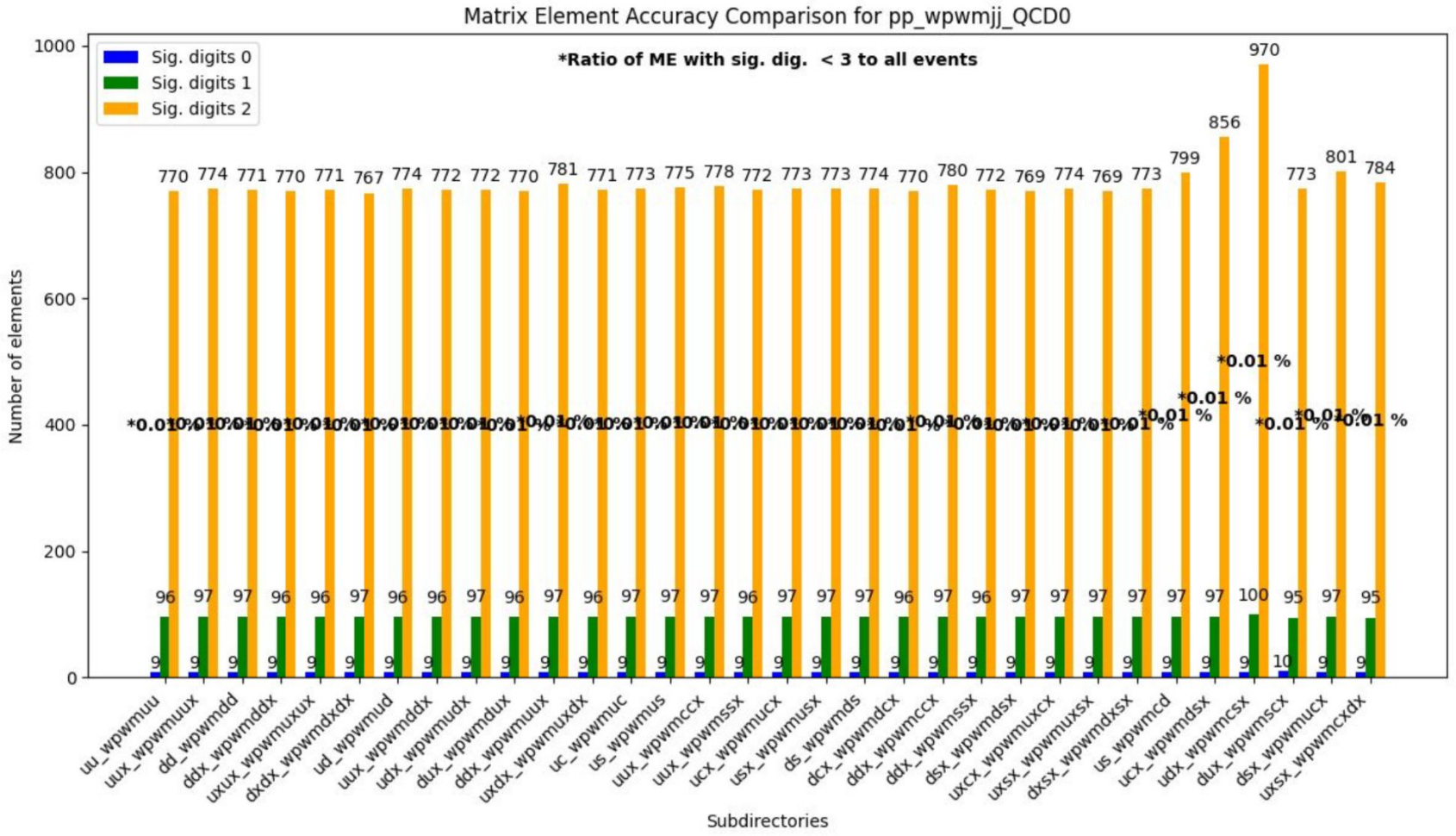




Backup

p p → W+ W- +2j QCD=0j

From 1.3% to <0.01%



Backup

gg -> ttxgg

```
Breakpoints: 5352595
operators:
+   : 3883352
-   : 1449172
*   : 9490
<   : 7616
+=  : 1652
-=  : 1308
pow : 5
```

CADNA does not support constexpr constants

-> replace by const fptype x = 68.f;

Math functions are overloaded by CADNA but we use std:: version

```
return std::sqrt(arg)
```

→

```
using std::sqrt
return sqrt(arg)
```

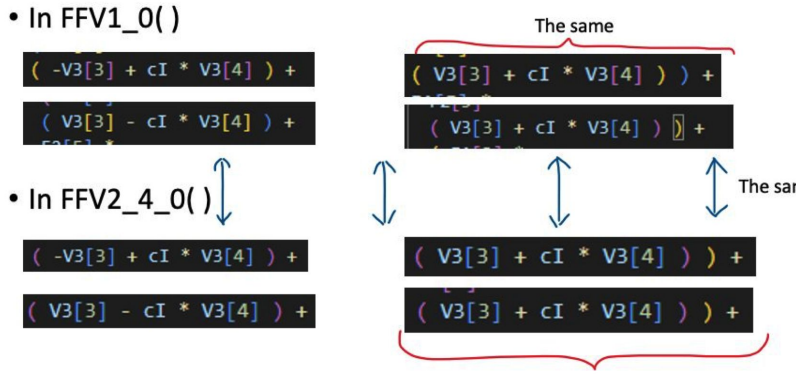
CADNA has multiple levels of checks performed. E.g. if only number of sig. dig. is up to interest, instead of all checks cadna_init(-1) (run time x100) only some cadna_init(M) or none cadna_init(0) (run time x10) can be performed.

Writing out of location of instability with gdb/lldb is slow (x500)

To make cadna erase the history simple static_cast is enough.

This can also easily introduce a bug but simple test is to worsen the precision and examine the propagation:

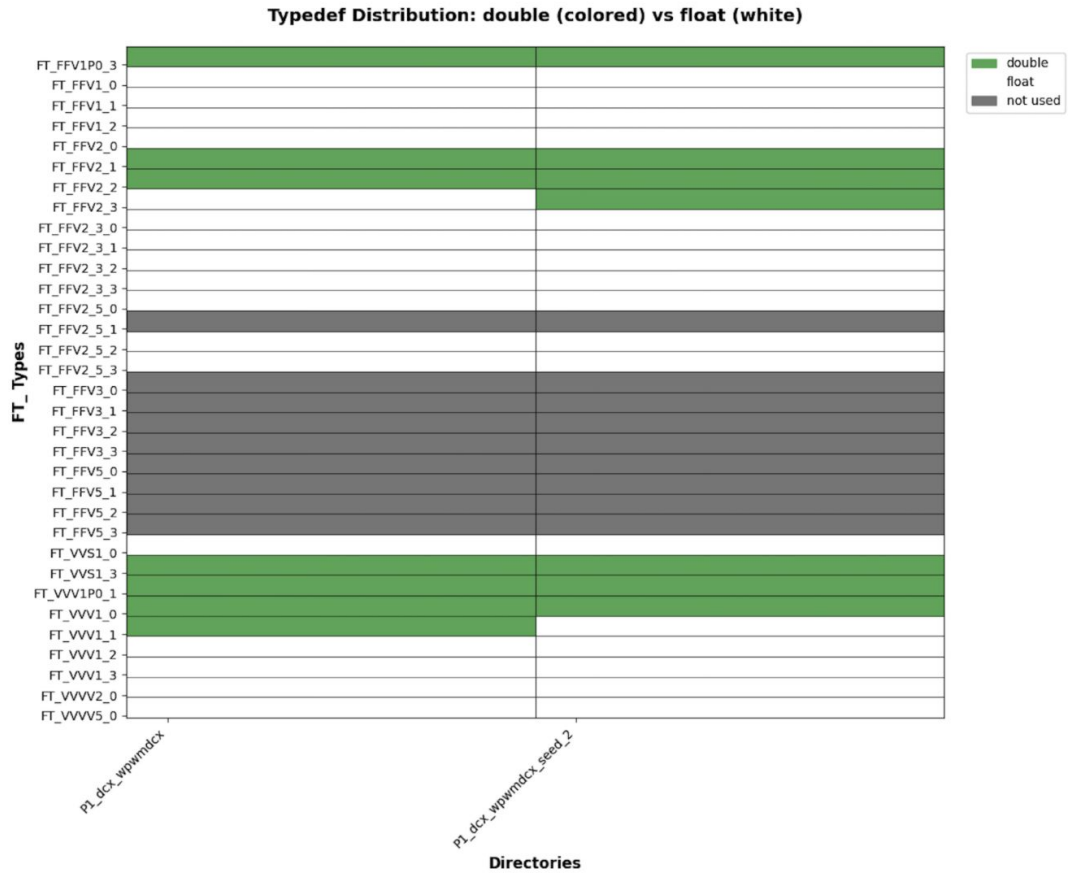
$$u.x_{worsen} = u.x - 8 \cdot 10^{log_{10}|u.x|-3}$$



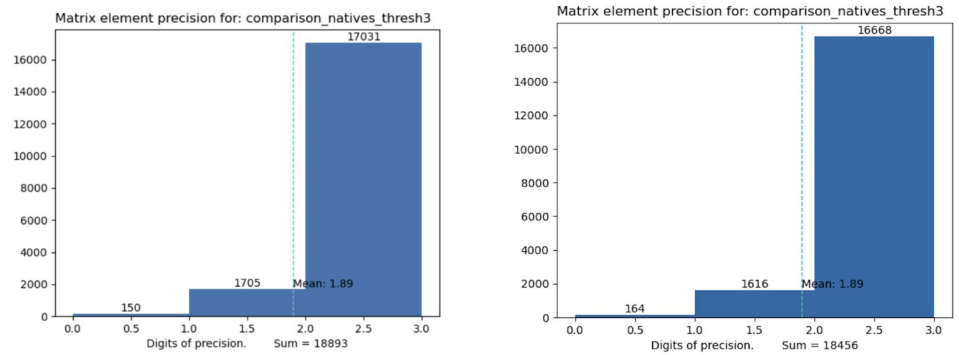
Thanks to Stephan Hageböck and Filip Optołowicz

Backup

seed 1 and 2 at 10^5



seed 1 and 2 at 10^7




```
It tooks 390.37s
👉55 compilations (0 failed) for 171.20s
👉55 executions   (0 failed) for 113.07s
{'double': {'fptype', 'VVV1P01', 'w', 'FFV12', 'oxxxxx', 'FFV11', 'vxxxxx'},
```

```
It tooks 352.49s
👉48 compilations (0 failed) for 151.25s
👉48 executions   (0 failed) for 104.18s
{'double': {'vxxxxx', 'fptype', 'FFV11', 'FFV12', 'VVV1P01', 'oxxxxx', 'w'},
```

Backup

Even quad is not correct

$$P = 333.75y^6 + x^2(11x^2y^2 - y^6 - 121y^4 - 2) + 5.5y^8 + x/(2y)$$
$$x = 77617; \quad y = 330996$$

float:	P =2.571784e+29
double:	P =1.17260394005318
quad:	P =1.172603940053178631858834904520 18
exact:	P ≈-0.827396059946821368141165095479816