Evaluating integrals and amplitudes with pySecDec

Vitaly Magerya

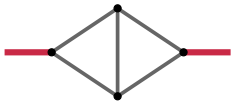
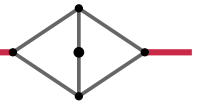
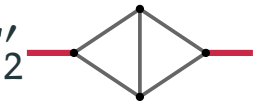
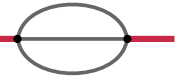
Together with G. Heinrich, S. P. Jones, M. Kerner, A. Olsson, and J. Schlenk.

Institute for Theoretical Physics,
Karlsruhe Institute of Technology

RADCOR 2023,
June 1, Crieff

Where does pySecDec fit in?

Basic steps of calculating scattering matrix elements:

1. Generate Feynman diagrams for the process.
   - $\mathscr{M} =$  $+ \ldots$
2. Project the diagrams onto scalar integrals.
   - $\mathscr{M} = C_1$  $+ C_2$  $+ C_3$  $+ C_4$  $+ \ldots$
3. Reduce the number of integrals using IBP relations.
   - $\mathscr{M} = C_1'$  $+ C_2'$  $+ C_3'$  $+ \ldots$
4. *Evaluate the loop integrals.*
   - Analytically: hard to impossible at two loops with multiple masses.
   - *Numerically:*
     - Numerical differential equations: DiffExp, AMFlow, SeaSyde, etc.
       [Talk by Xiao Liu]
     - Mellin-Barnes representation: MB, Ambre, etc.
     - Tropical sampling: Feyntrop. [Talks by Henrik Munch and Michael Borinsky]
     - *Sector decomposition:* pySecDec, Fiesta, sector_decomposition, etc.
5. Integrate the matrix element over kinematics.

What is pySecDec?

pySecDec: library for *numerically evaluating parametric integrals* via sector decomposition and Monte Carlo integration. [Heinrich et al '23, '21, '18, '17]

- `https://github.com/gudrunhe/secdec`
  - Installable via `python3 -m pip install pySecDec`.
- Written in *Python*, C++, Form; works on CPUs and GPUs.
- Multiple sector decomposition methods: iterative, geometric.
- Multiple integration algorithms:
  - Best: *Randomized Quasi Monte Carlo* (QMC); [Borowka et al '18]
  - Classic: Vegas/Suave/Divonne/Cuhre (Cuba), Cquad (Gsl).
- Adaptive evaluation of *weighted sums of integrals* (i.e. amplitudes).
- Builtin *asymptotic expansion of integrals* (expansion by regions).

New release 1.6 comes with:

- New evaluator "Disteval": *3x-9x faster*, distributed.
- Support for big sum coefficients (e.g. IBP coefficients).
- Unlimited lucky QMC lattices via the new "median lattice" construction.
- Automatic minimal extra regulators for expansion by regions.

$\Longrightarrow$ Available now (arXiv:2305.19768 out this morning).

Sector decomposition in short

$$I = \int_0^1 \mathrm{d}x \int_0^1 \mathrm{d}y \left(x+y\right)^{-2+\varepsilon} = ?$$

Problem: the integrand diverges at $x, y \longrightarrow 0$, can't integrate numerically.

Solution: [Heinrich '08; Binoth, Heinrich '00]

1. Factorize the divergence in x and y with sector decomposition:

   * $I = \int \cdots \times \underbrace{\left(\theta\left(x>y\right)\right.}_{\text{Sector 1}} + \underbrace{\left.\theta\left(y>x\right)\right)}_{\text{Sector 2}} = \int_0^1 \mathrm{d}x \int_0^x \mathrm{d}y \left(x+y\right)^{-2+\varepsilon} + \begin{pmatrix} x \\ \updownarrow \\ y \end{pmatrix}$

2. Rescale the integration region in each sector back to a hypercube:

   * $I \overset{y \to x\,y}{=} \int_0^1 \mathrm{d}x \underbrace{x^{-1+\varepsilon}}_{\text{Factorized pole}} \int_0^1 \mathrm{d}y \left(1+y\right)^{-2+\varepsilon} + \begin{pmatrix} x \\ \updownarrow \\ y \end{pmatrix}$

3. Extract the pole at $x \longrightarrow 0$ analytically, expand in ε:

   * $I = -\frac{2}{\varepsilon} \int_0^1 \mathrm{d}y \left(1+y\right)^{-2+\varepsilon} = -\frac{2}{\varepsilon} \int_0^1 \mathrm{d}y \left(\frac{1}{\left(1+y\right)^2} - \frac{\ln\left(1+y\right)}{\left(1+y\right)^2}\varepsilon + \mathscr{O}\left(\varepsilon^2\right)\right)$

4. Integrate each term in ε numerically (they all converge now).

In practice: geometric sector decomposition. [Bogner, Weinzierl '07; Kaneko, Ueda, '09]

Contour deformation in short

$$I \equiv \int \mathrm{d}^n\vec{x}\, \frac{U^\alpha\left(\vec{x}\right)}{F^\beta\left(\vec{x}, \dots\right) + i0}$$

Problem: can't integrate numerically if $F = 0$ inside the integration region.

Solution: *deform $\vec{x}$ into the complex plane* to escape the pole:

$$\vec{x} \to \vec{x} + i\,\vec{\Delta}\left(\vec{x}\right)$$

$$\Rightarrow \begin{cases} F \to F + i\,\Delta\,\partial_x F - \Delta^2\,\partial_x^2 F - i\,\Delta^3\,\partial_x^3 F + \mathscr{O}\left(\Delta^4\right), \\ \mathrm{Im}\,F \to \Delta\,\partial_x F - \Delta^3\,\partial_x^3 F + \mathscr{O}\left(\Delta^5\right). \end{cases}$$

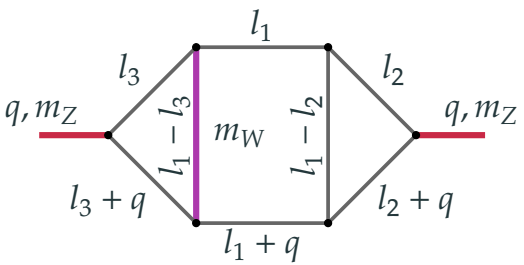
Choose $\vec{\Delta}\left(\vec{x}\right)$ to enforce the $+i0$ prescription ($\mathrm{Im}\,F > 0$):

$$\vec{\Delta}\left(\vec{x}\right) = \lambda\,\vec{\partial}_x F\left(\vec{x}\right) \quad \Rightarrow \quad \mathrm{Im}\,F \approx \lambda\,\left(\partial_x F\right)^2 - \lambda^3\left(\partial_x F\right)^3\,\partial_x^3 F + \mathscr{O}\left(\lambda^5\right) > 0.$$

- Lambda should be small enough that $\mathrm{Im}\,F > 0$.
- But: larger λ improves convergence (the pole is further away).
- In practice: choose λ heuristically, but decrease it if $\mathrm{Im}\,F < 0$.
  - Gradient-based λ optimization can be useful. [Winterhalder et al '21]

Usage & Performance

Using pySecDec for single Feynman integrals



$= ?$

Generate the integration library:

```
import pySecDec as psd
if __name__ == "__main__":
    psd.loop_package(
        name="dia1",
        loop_integral=psd.LoopIntegralFromPropagators(
            loop_momenta=["l1", "l2", "l3"],
            propagators=[
                "(l1)**2", "(l2)**2", "(l3)**2",
                "(l1+q)**2", "(l2+q)**2",
                "(l3+q)**2", "(l1-l2)**2",
                "(l1-l3)**2-mw2"
            ],
            replacement_rules=[("q*q", "mz2")]),
        real_parameters=["mz2", "mw2"],
        requested_orders=[0])
```

With the *old integrator* "IntLib":

Compile the library:

```
$ cd dia1 && make SECDEC_WITH_CUDA_FLAGS="-arch=sm_80"
```

Run it:

```
from pySecDec.integral_interface import IntegralLibrary
lib = IntegralLibrary("dia1/dia1_pylink.so")
lib.use_Qmc(verbosity=1)
mz2, mw2 = 1.0, 0.78
_,_,result = lib([mz2,mw2], epsrel=1e-7, verbose=True)
print(result)
```

With the *new integrator* "Disteval":

Compile the library:

```
$ cd dia1 && make disteval SECDEC_WITH_CUDA_FLAGS="-arch=sm_80"
```

Run it:

```
from pySecDec.integral_interface import DistevalLibrary
lib = DistevalLibrary("dia1/disteval/dia1.json")
result = lib(parameters={"mz2": 1.0, "mw2": 0.78}, epsrel=1e-7)
print(result)
```

The result, a numeric series in ε:

```
+ ((6.82645729748523067e+00,-1.60711801420445148e+01)
+/- (8.07205205949069617e-07,7.55815020343424309e-07))
+ O(eps)
```

Using pySecDec for amplitudes

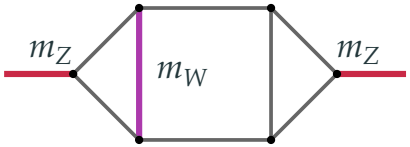
Generate the integration library:

```
import pySecDec as psd
if __name__ == "__main__":
    # First integral
    int1 = psd.LoopPackage(
        name="integral1",
        loop_integral=psd.LoopIntegralFromPropagators(...),
        real_parameters=[...])
    # Second integral
    int2 = psd.LoopPackage(...)
    ...
    # The sums / amplitudes
    psd.sum_package('amplitudes',
        [int1, int2, ...],
        regulators=['eps'],
        requested_orders=[0],
        coefficients={
            "sum one": ["(2+eps^3)/(-4+5*eps) ...", ...],
            "sum two": [...],
            ...},
        real_parameters=[...])
```

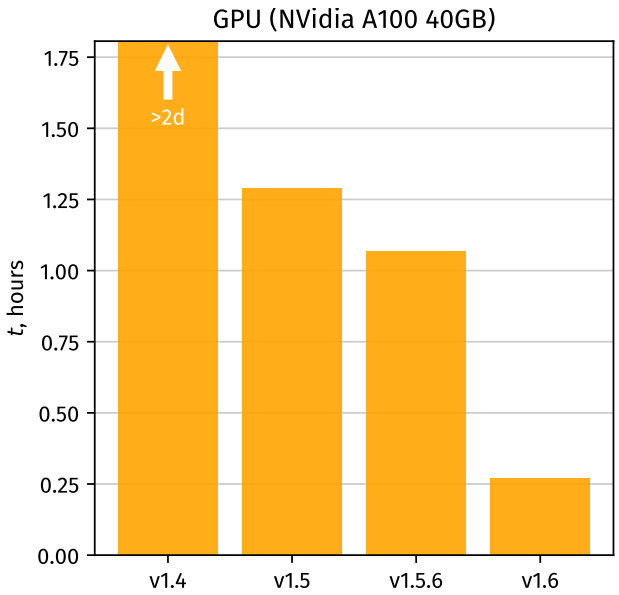
Compile an *run*: same as for a single integral.

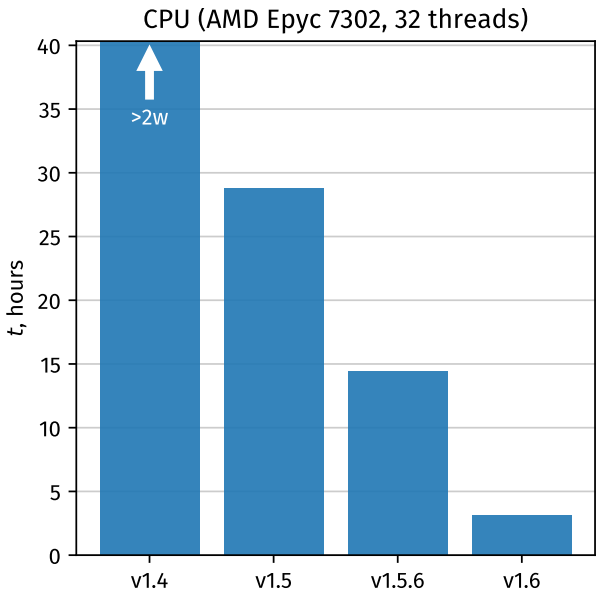
- Integrals in *sums are sampled adaptively*, optimizing for fastest time to achieve the requested sum precision (since v1.5).
- Sum *coefficients can be arbitrary rational expressions* (since v1.6); they will be evaluated with infinite precision during integration.
  - Big coefficient expression coming from IBP reduction are no problem.
- Under the hood single integrals are implemented as sums of sectors.

Performance improvements by pySecDec version

Time to integrate 

to 7 digits of precision with pySecDec:

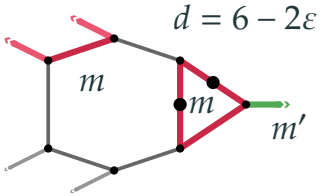


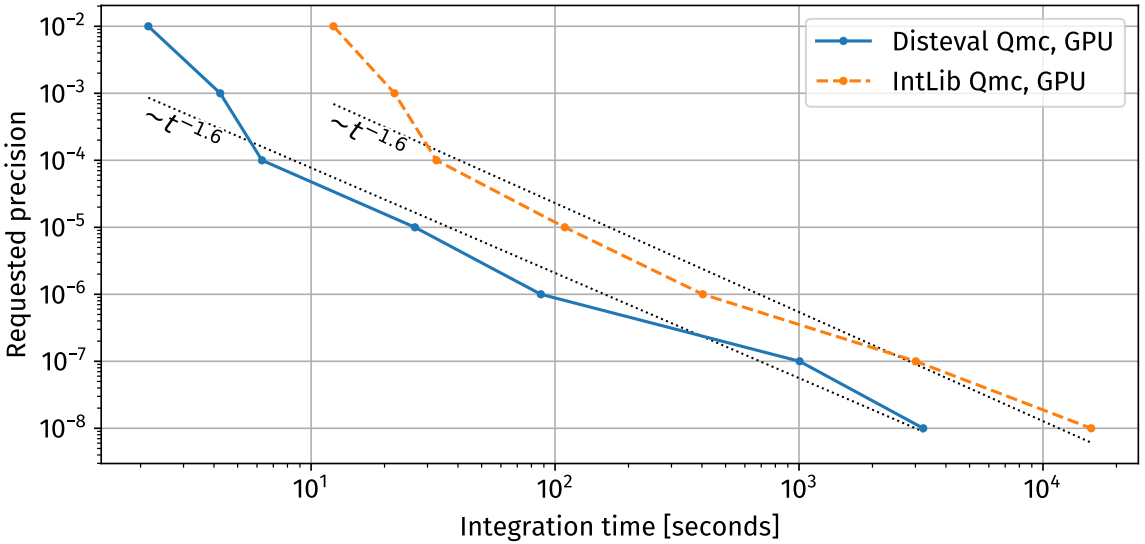


Speedup sources:

- *v1.5:* adaptive sum sampling, auto contour deformation adjustment;
- *v1.5.6:* microoptimizations in the integrand code;
- *v1.6:* a new Quasi-Monte-Carlo integrator "DISTEVAL":
  - same algorithm as the old integrator ("INTLIB"), faster implementation;
  - CPU & GPU: fusion of the integration code with the integrand code;
  - CPU: better processor utilization via SIMD intstructions (AVX2, FMA);
  - GPU: result summation directly on the GPU, minimized synchronization.

Integration time: DISTEVAL vs INTLIB



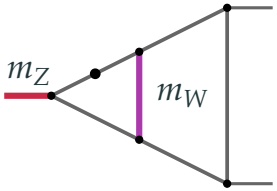
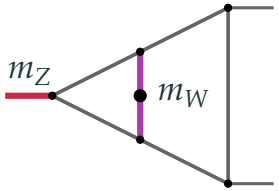
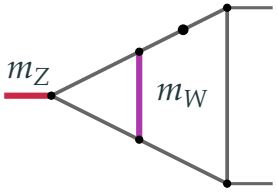
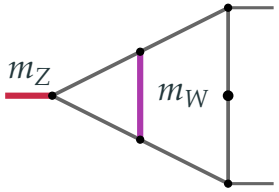


Integrator \ Accuracy		10^{-3}	10^{-4}	10^{-5}	10^{-6}	10^{-7}	10^{-8}
GPU	DISTEVAL	4.2 s	6.3 s	27 s	1.5 m	17 m	54 m
	INTLIB	22.0 s	22.0 s	110 s	6.7 m	50 m	263 m
	Speedup	5.2	5.2	4.1	5.6	3.0	4.9
CPU	DISTEVAL	5.1 s	14 s	1.6 m	8.3 m	57 m	4.7 h
	INTLIB	20.8 s	86 s	14.2 m	62.2 m	480 m	43.1 h
	Speedup	4.1	6.1	8.7	7.5	8.4	9.2

[GPU: NVidia A100 40GB; CPU: AMD EPYC 7F32 with 32 threads]

On the choice of integrals

Integration time of similarly looking integrals to 10^{-3} precision:[1]

	orders	t, s		orders	t, s
	$\varepsilon^{-3} \dots \varepsilon^{0}$	27		$\varepsilon^{-2} \dots \varepsilon^{0}$	57
	$\varepsilon^{-2} \dots \varepsilon^{0}$	1230		$\varepsilon^{-2} \dots \varepsilon^{0}$	>9000

Takeaway: for best performance, test the integration speed and adjust the selection of the master integrals.

[1] pySecDec 1.5.3, NVidia A100 GPU.

Distributed evaluation with DISTEVAL

DISTEVAL splits the integration between the *main* and the *worker* processes.

- Integrands are evaluated on the workers.
- The main↔worker communication is done via byte streams (pipes).

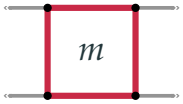
⇒ The *workers can run on remote machines*, as long as a pipe to it can be established (e.g. if you `ssh` into it).

For example, to use one GPU on `machine1` and two GPUs on `machine2`:

```
from pySecDec.integral_interface import DistevalLibrary
lib = DistevalLibrary("dia1/disteval/dia1.json",
    workers=[
        "ssh machine1 python3 -m pySecDecContrib pysecdec_cudaworker -d 0",
        "ssh machine2 python3 -m pySecDecContrib pysecdec_cudaworker -d 0",
        "ssh machine2 python3 -m pySecDecContrib pysecdec_cudaworker -d 1"
    ])
result = lib(parameters={"mz2": 1.0, "mw2": 0.78}, epsrel=1e-7)
print(result)
```

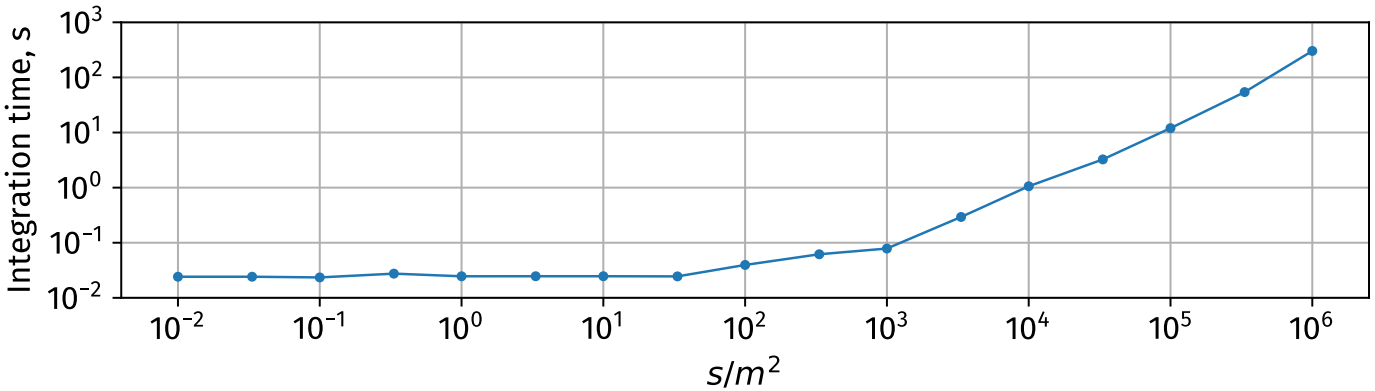
Asymptotic expansion of integrals

Asymptotic expansion: the motivation

$I(m^2, s, t) \equiv$ 

$\sim \int \mathrm{d}x_1 \cdots \mathrm{d}x_4 \frac{U^{2\varepsilon}(\vec{x})}{F^{2+\varepsilon}(\vec{x}, m^2, s, t)} \delta(1 - x_{1234}) = ?$

Problem: when $s \gg m^2$ numerical integration converges poorly.
E.g. the time to integrate I to 10^{-3} accuracy:[2]



Solution: take out the small ratio (m^2/s) from the integrand via *asymptotic expansion* ("expansion by regions"):

$$I = \left(\int \cdots \text{no } m^2/s \text{ here} \cdots\right)\left(\frac{m^2}{s}\right)^{-\varepsilon} + \left(\int \cdots\right)\left(\frac{m^2}{s}\right)^{0} + \left(\int \cdots\right)\left(\frac{m^2}{s}\right)^{1-\varepsilon} + \cdots.$$

[2] pySecDec v1.6 Disteval, NVidia A100 GPU, $s = 1$, $t = -2$.

Asymptotic expansion: the method

Method of *expansion by regions*: [Beneke, Smirnov '98]

- Split the integration space into *regions*, ordering the variables by the smallness parameter (m^2/s) in each.
  - E.g. $\{x_1, \dots, x_4\} \sim \left(\frac{m^2}{s}\right)^{\{-1,-1,0,0\}}$, $\sim \left(\frac{m^2}{s}\right)^{\{-1,0,0,-1\}}$, etc.
  - The required regions can be found geometrically from the Newton polytope of $F + U$. [Jantzen '11; talk by Yao Ma]
- Taylor-expand the integrand in each region (in m^2/s).
- Integrate and sum the regions.
  - OK to integrate each region's integrand over the full integration domain.

Implementations:

- ASY2.M (part of FIESTA); [Jantzen, Smirnov, Smirnov '12]
- ASPIRE; [Ananthanarayan, Pal, Ramanan, Sarkar '18]
- pySECDEC v1.5 (via `pySecDec.loop_regions`). [Heinrich et al '21]

Asymptotic expansion and extra regulators

$I\left(m^2, s, t\right) =$ 

$= \int \mathrm{d}x_{1234}\, U^{\alpha}\left(\vec{x}\right) F^{\beta}\left(\vec{x}, m^2, s, t\right) \delta(1 - x_{1234})$

Expansion by regions of I in small m^2/s introduces *spurious singularities* not regulated by the dimensional regularization.

$\Longrightarrow$ *Extra regulators* need to be introduced, e.g.: [Smirnov '97]

$$\mathrm{ebr}[\, I \,] = \lim_{\nu_{1,2,3,4} \to 0} \mathrm{ebr}\left[\int \mathrm{d}x_{1234}\, U^{\alpha}\, F^{\beta}\, \delta(1 - x_{1234}) \cdot x_1^{\nu_1} x_2^{\nu_2} x_3^{\nu_3} x_4^{\nu_4}\right]$$

Many valid options:

* each ν_i as an independent regulator variables;
* one variable n and $\nu_{1,2,3,4} = \{n,\, n/2,\, n/3,\, n/5\}$; etc.

Is a more *compact construction* possible? Yes. [Heinrich et al '21; Schlenk '16]

* A single regulator variable is always enough.
* Sufficient conditions via `extra_regulator_constraints()`:

$$\nu_2 - \nu_4 \neq 0 \quad \text{and} \quad \nu_1 - \nu_3 \neq 0.$$

* A solution via `suggested_extra_regulator_exponent()`:

$$\nu_{1,2,3,4} = \{0,\, 0,\, n,\, -n\}\,.$$

Using pySecDec with asymptotic expansion

No extra regulators: sometimes OK, but will fail for this example.

```
import pySecDec as psd
if __name__ == "__main__":
    # define the integral
    loopint = \
        psd.LoopIntegralFromPropagators(
            loop_momenta=["l"],
            propagators=[
                "(l)^2-mm",
                "(l-p1)^2-mm",
                "(l-p1-p2)^2-mm",
                "(l-p1-p2-p3)^2-mm"],
            replacement_rules = [
                ("p1*p1", "0"),
                ("p2*p2", "0"),
                ("p3*p3", "0"),
                ("p1*p2", "s/2"),
                ("p1*p3", "t/2"),
                ("p2*p3", "-s/2-t/2")])
    # expand by regions up to O(mm)
    terms = psd.loop_regions(
        name="box",
        loop_integral=loopint,
        smallness_parameter="mm",
        expansion_by_regions_order=0)

    # generate the library (WILL FAIL HERE)
    psd.sum_package("box_expansion",
        terms,
        real_parameters=["s", "t", "mm"],
        regulators=["eps"],
        requested_orders=[0])
```

Automatic minimal extra regulator, new in v1.6.

```
import pySecDec as psd
if __name__ == "__main__":
    # define the integral
    loopint = \
        psd.LoopIntegralFromPropagators(
            loop_momenta=["l"],
            propagators=[
                "(l)^2-mm",
                "(l-p1)^2-mm",
                "(l-p1-p2)^2-mm",
                "(l-p1-p2-p3)^2-mm"],
            replacement_rules = [
                ("p1*p1", "0"),
                ("p2*p2", "0"),
                ("p3*p3", "0"),
                ("p1*p2", "s/2"),
                ("p1*p3", "t/2"),
                ("p2*p3", "-s/2-t/2")])
    # expand by regions up to O(mm)
    terms = psd.loop_regions(
        name="box",
        loop_integral=loopint,
        smallness_parameter="mm",
        expansion_by_regions_order=0,
        extra_regulator_name="n")
    # generate the library
    psd.sum_package("box_expansion",
        terms,
        real_parameters=["s", "t", "mm"],
        regulators=["n", "eps"],
        requested_orders=[0, 0])
```

QMC integration & Lattices

Randomized Quasi Monte Carlo integration

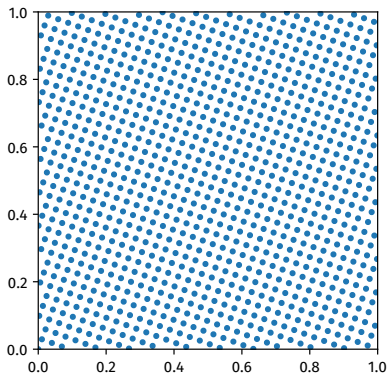
To numerically estimate an n-dimensional integral

$$I \equiv \int_0^1 \mathrm{d}^n\vec{x}\, I\left(\vec{x}\right):$$

1. Use a *low-discrepancy sequence* of size N, e.g. a *rank-1 lattice*:

$$\vec{x}_i^{(k)} = \left(\frac{i \cdot \vec{g}_N}{N} + \vec{\Delta}^{(k)}\right) \bmod 1,$$

   where
   - $\vec{g}_N \in \mathbb{N}^n$ are the *generating vectors*,
   - and $\vec{\Delta} \in \mathrm{Uniform}(0;1)^n$ is a *random shift*,
   - $i = 1 \ldots N$ points,
   - $k = 1 \ldots K$ shifts.

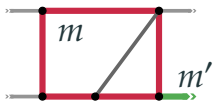


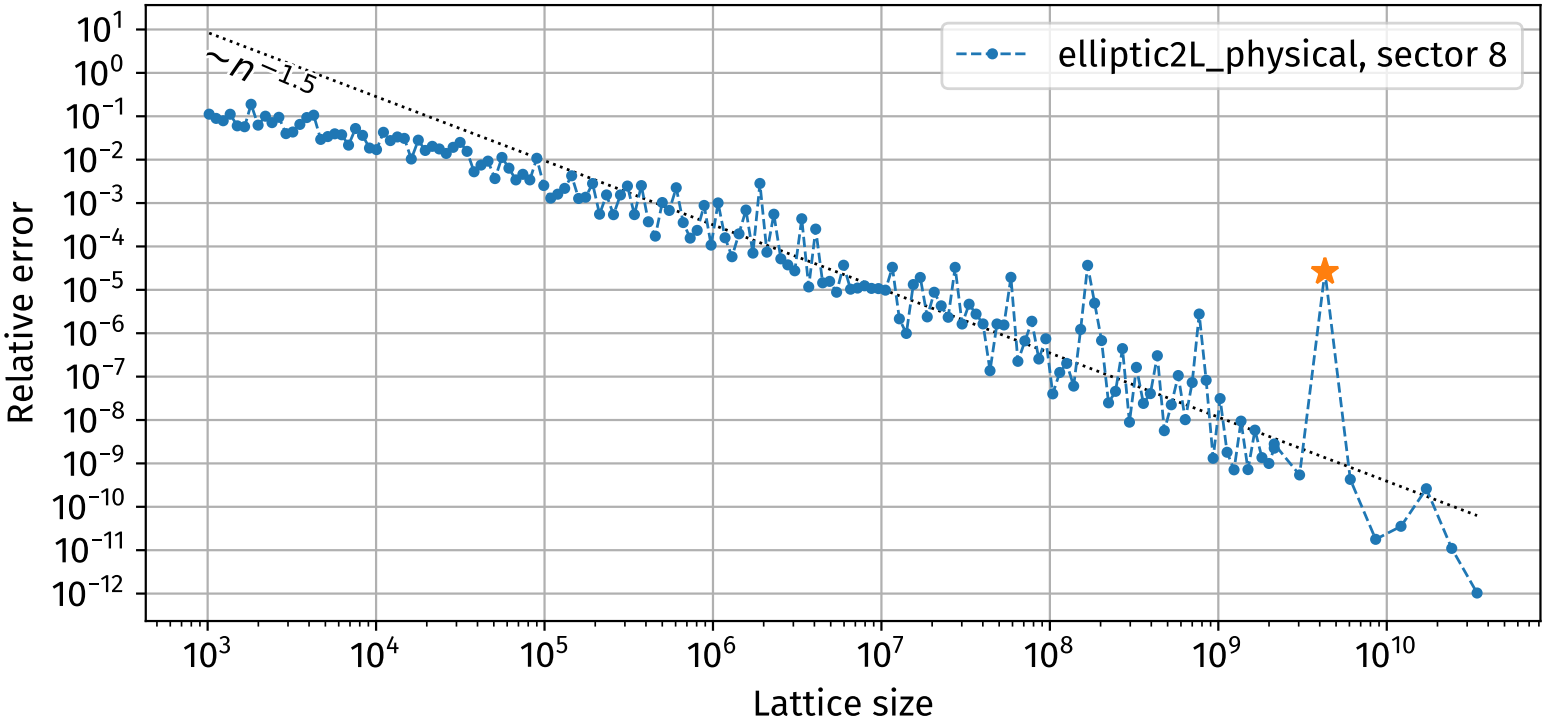
2. Estimate I as $\mathrm{mean}_k\, \mathrm{mean}_i\, I\left(\vec{x}_i^{(k)}\right)$.
3. Estimate the error as $\mathrm{stdev}_k\, \mathrm{mean}_i\, I\left(\vec{x}_i^{(k)}\right)/\sqrt{K-1}$.

There exist $\vec{g}_N$ that guarantee error scaling of $\sim 1/N^\alpha$, if $\partial_x^{(\alpha)} I\left(\vec{x}\right)$ is square-integrable and periodic. [Dick, Kuo, Sloan '13]

$\Longrightarrow$ "Fast component-by-component" (CBC) construction. [Nuyens, Cools '06]

QMC error scaling by lattice size

Precision by lattice for a 2-loop massive box: 



Asymptotic error scaling is $\sim 1/N^{1.5}$, but:

- some lattices are *unlucky*, sometimes by multiple orders of magnitude;
- the CBC lattice construction needs $\mathscr{O}(N)$ bytes of memory;
  - $\Longrightarrow$ we are able to construct lattices only up to $4{\cdot}10^{10}$ on a 2TB server.

N.B.: plain Monte Carlo would have scaled as $\sim 1/N^{0.5}$.

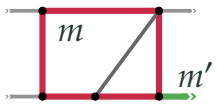
Median lattices

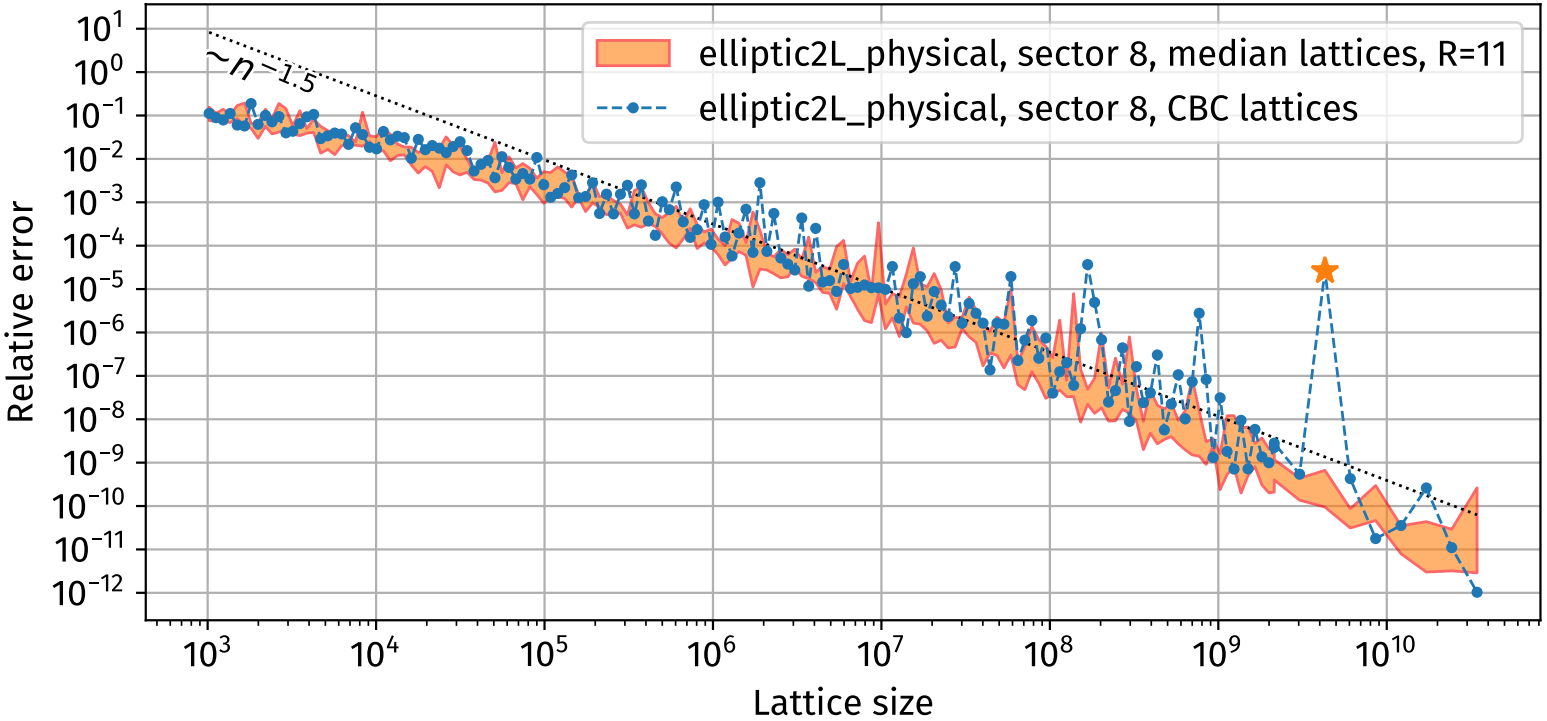
Question: how to choose $\vec{g}_N$ for large N using limited memory and avoid unlucky lattices for a given integrand?
Observation: many randomly chosen lattices are quite good.
Solution: [Goda, L'Ecuyer '22]

1. Choose R *random* generating vectors $\vec{g}_N \in \mathrm{Uniform}(0; N-1)$.
2. Estimate the integral on each of the corresponding lattices.
3. Choose the lattice corresponding to the *median* integral value.

$\Longrightarrow$ With *high probability the error will scale almost optimally*.
More precisely: for any $0 < \varepsilon$ and $0 < \rho < 1$, if the integrand $I(x)$ is periodic and has square-integrable $\partial_x^{(\alpha)} I(x)$, the integration error will be $C(\alpha, \varepsilon)/(\rho N)^{\alpha-\varepsilon}$ with probability $1 - \rho^{(R+1)/2}/4$.

Median lattices in practice

Precision by lattice for a 2-loop massive box: 



In short:

- median lattices are on average competitive with CBC lattices;
- at higher precisions the worst unlucky lattices are avoided;
- no limitation on the lattice size;
- but: need to be constructed on the fly, interleaved with integration.

Summary

Summary

pySecDec provides:

- Numerical evaluation of *massive & massless multi-loop integrals*.
- Optimized evaluation of *weighted sums of integrals* (i.e. amplitudes).
  - Proven in multiple 2-loop calculations.

    [Talk by Bakul Agarwal; Chen et al '20, '19; Jones, Kerner, Luisoni '18]
  - Automatically applicable to single integrals (sums of sectors) too.
- *Asymptotic expansion* (by regions) of integrals.

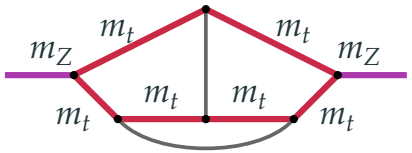
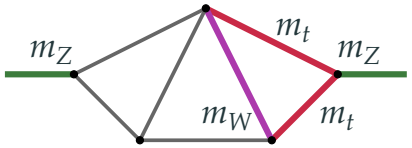
The latest release v1.6 includes: [Heinrich et al '23]

- New and *faster Quasi Monte Carlo evaluator* "Disteval":
  - 3x-5x performance improvement on a GPU, 4x-9x on a CPU;
  - optional distributed evaluation.
- QMC *lattices of unlimited size* due to the "median lattice" construction.
  - For both Disteval and the old Qmc evaluator.
- Minimal *extra regulator construction* for expansion by regions.
- Support for arbitrary rational coefficients in sums.

Backup slides

Expected performance for 3-loop EW integrals

pySecDec Disteval *integration times* for 3-loop self-energy integrals:[3]

Diagram \ Relative precision		10^{-3}	10^{-4}	10^{-5}	10^{-6}	10^{-7}	10^{-8}
	GPU	15s	20s	40s	200s	13m	50m
	CPU	10s	50s	400s	4000s	180m	1200m
	GPU	18s	19s	30s	20s	1.2m	2m
	CPU	5s	14s	60s	50s	12m	16m
	GPU	6s	11s	12s	30s	3m	24m
	CPU	5s	10s	50s	800s	60m	800m

[Same diagrams as in Dubovyk, Usovitsch, Grzanka '21]

In short: *seconds to minutes per integral* to achieve practical precision.

[3] *GPU*: NVidia A100 40GB; *CPU*: AMD Epyc 7302 with 32 threads.

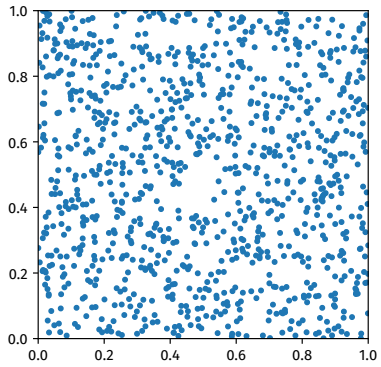
Monte Carlo Integration

To numerically approximate an n-dimensional integral

$$I \equiv \int_0^1 \mathrm{d}^n\vec{x}\, I\left(\vec{x}\right),$$

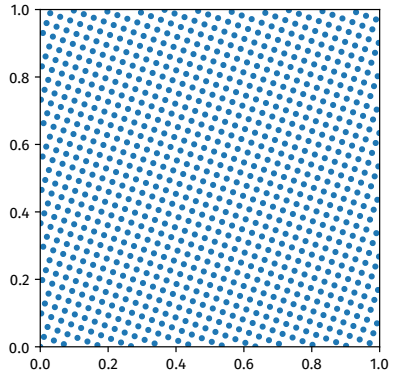
Monte Carlo integration:

1. Sample $\vec{x}_{1\ldots N}$ uniformly from $[0;1)^n$.
2. Estimate I as $\mathrm{mean}\left(I\left(\vec{x}_i\right)\right)$.
3. Estimation error is $\sqrt{\frac{1}{N}\,\mathrm{stdev}\left(I\left(\vec{x}_i\right)\right)} \sim N^{-\frac{1}{2}}$.



In pySecDec: Vegas/Suave/Divonne integrators via Cuba. [Hahn '04]

Monte Carlo, Quasi



Quasi-Monte-Carlo integration:

1. Let $\vec{x}_{1\ldots N}$ be a *low-discrepancy sequence* in $[0;1)^n$.
2. Estimate I as $\mathrm{mean}\left(I\left(\vec{x}_i\right)\right)$.
3. Error is $V(I)\,D\left(\vec{x}_i\right)$, where
   - V is the *Hardy-Krause variation* of I (impractical to calculate),
   - and D is the *star discrepancy* of $\left\{\vec{x}_1, \ldots, \vec{x}_N\right\}$,

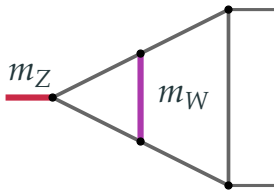
$$D\left(\vec{x}_i\right) \equiv \max_{Q\subset[0;1)^n} \left| \frac{\text{number of } \vec{x}_i \in Q}{N} - \text{volume of } Q \right|.$$

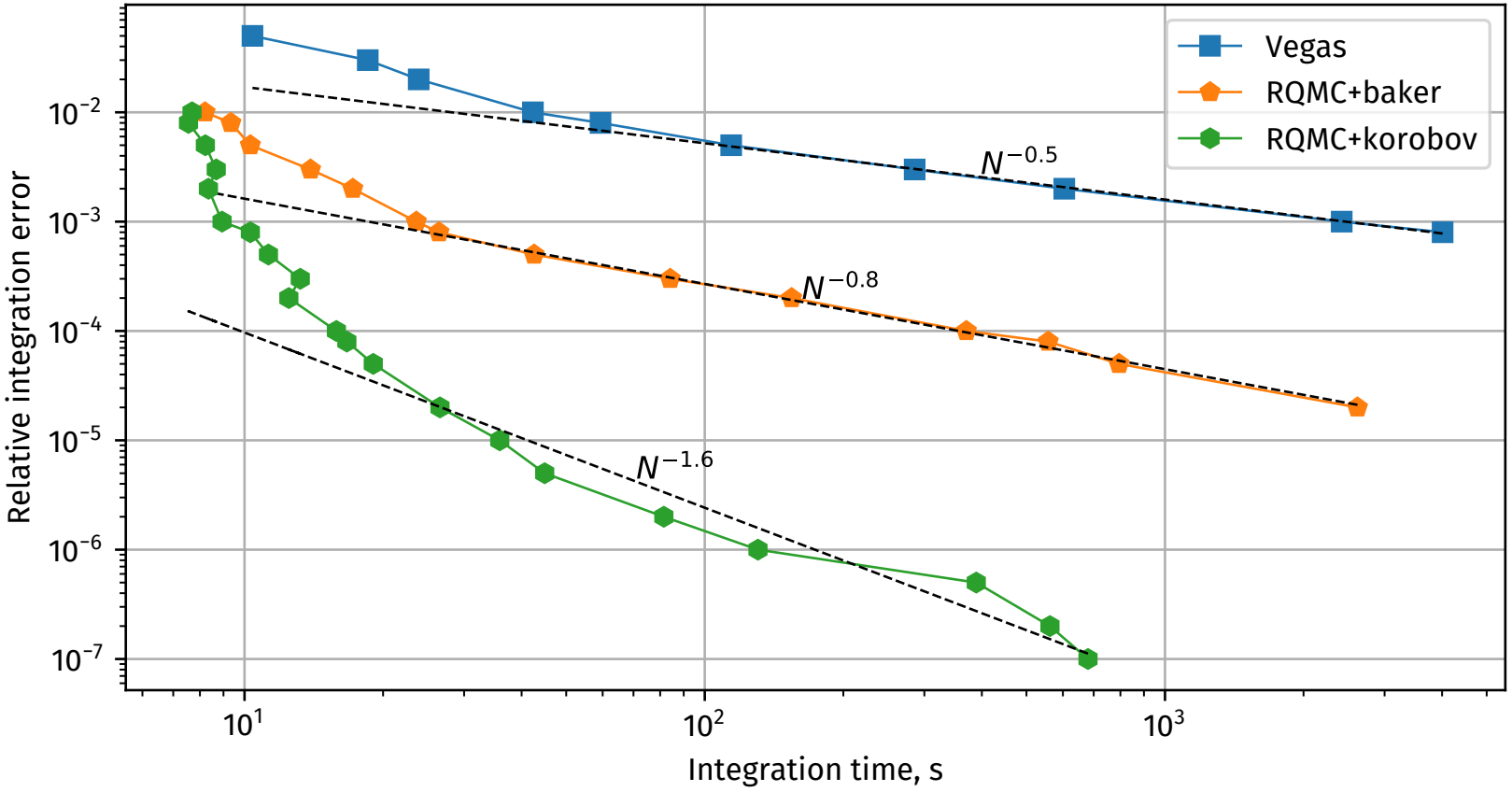
Low-discrepancy sequences: [Dick, Kuo, Sloan '13]

- digital sequences (Sobol', Faure, etc), with error $\sim N^{-1}$;
- Owen's scrambled nets, with error $\sim N^{-1.5}$ if $\partial_x I$ is square-integrable;
- lattice rules, with error $\sim N^{-\alpha}$, if $\partial_x^{(\alpha)} I$ is square-integrable and periodic.
  - To enforce periodicity, use a transformation ($x \longrightarrow y$):
    - baker's, $y(x) = 1 - |2x - 1|$;
    - Korobov, $\mathrm{d}y/\mathrm{d}x \sim x^l\,(1-x)^r$; etc.

Monte Carlo vs RQMC

Integration time scaling for Monte Carlo (VEGAS) vs Randomized Quasi Monte Carlo (QMC).[4]





[4] pySECDEC v1.5.3 on NVidia A100 GPU.

Integration library generated code: v1.5 vs v1.6

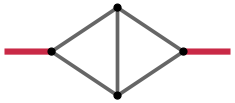
pySecDec *v1.5* & Qmc:

```
integrand_return_t sector_1_order_0_integrand(
        const real_t * const integration_variables,
        const real_t * const realp,
        const complex_t * const complexp,
        const real_t * const deformp,
        secdecutil::ResultInfo * const result_info) {
    real_t mz2 = realp[0], mw2 = realp[1];
    real_t x0 = integration_variables[0];
    [...]
    integrand_return_t tmp[49];
    tmp[1] = x2*x1;
    tmp[2] = tmp[1] + x1;
    tmp[3] = tmp[2]*mz2;
    tmp[4] = mz2*x2;
    [...]
    return tmp[0];
};
[...]
template<...> void compute(
        const U index0,
        const std::vector<U>& genvec,
        const std::vector<D>& shift,
        T* results,
        const U r_size_over_m,
        const U stride, const U latsize, const U nshifts,
        I& func) {
    for (U k = 0; k < nshifts; k++) {
        for (U i = index0; i < latsize; i += stride) {
            std::vector<D> x(func.dimension, 0);
            for (U j = 0; j < func.dimension; j++) {
                x[j] = modf(
                    mul_mod(i, genvec[j], latsize)/latsize
                        + shift[k*func.dimension+j]);
            }
            T point = func(x.data());
            results[k*r_size_over_m] += point;
} } }
```

pySecDec *v1.6* & Disteval:

```
int integral__sector_1_order_0(
        result_t * restrict result,
        uint64_t latsize,
        uint64_t index1, uint64_t index2,
        const uint64_t * restrict genvec,
        const real_t * restrict shift,
        const real_t * restrict realp,
        const complex_t * restrict complexp,
        const real_t * restrict deformp) {
    const real_t mz2 = realp[0];
    const real_t mw2 = realp[1];
    [...]
    resultvec_t sum = {0, 0, 0, 0};
    int64_t latidx_x0 = mulmod(genvec[0], index, latsize);
    [...]
    for (uint64_t i = index1; i < index2; i += 4) {
        int64_t li_x0_0 = latidx_x0;
        latidx_x0 = modonce(latidx_x0 + genvec[0], latsize);
        [...]
        realvec_t x0 = {{ li_x0_0, li_x0_1, li_x0_2, li_x0_3 }};
        x0 = modonce(x0*(1.0/latsize) + shift[0], 1);
        [...]
        realvec_t jac = korobov3_w(x0) * korobov3_w(x1) * [...];
        [...]
        x0 = korobov3_f(x0);
        [...]
        auto tmp1_1 = x1*mw2;
        auto tmp1_2 = 2*tmp1_1;
        auto tmp1_3 = tmp1_2 + mw2;
        auto tmp1_4 = tmp1_3*x3;
        auto tmp1_5 = tmp1_4 + mw2;
        [...]
        sum = sum + jac*(tmp3_356);
    }
    *result = componentsum(sum);
    return 0;
}
```

Adaptive sampling of amplitudes

Amplitude term	Naive sampling	Naive error	Better sampling	Better error
1 	10^6 samples	$1 \cdot 10^{-6}$	$\frac{1}{2} \cdot 10^6$ samples	$2 \cdot 10^{-6}$
10 	10^6 samples	$10 \cdot 10^{-6}$	$\frac{1}{2} \cdot 10^6$ samples	$20 \cdot 10^{-6}$
50 	10^6 samples	$50 \cdot 10^{-6}$	$2 \cdot 10^6$ samples	$25 \cdot 10^{-6}$
Total:	$3 \cdot 10^6$	$51 \cdot 10^{-6}$	$3 \cdot 10^6$	$32 \cdot 10^{-6}$

[Example assumes integration error $= 1/n$]

pySecDec now automatically optimizes the total integration time based on

- how fast each integral can be sampled,
- how well it converges,
- how large its coefficient is.

$\Rightarrow$ Automatic *speedup for amplitudes* (weighted sums of integrals).

$\Rightarrow$ Automatic speedup for single integrals too (sums of sectors).

$\Rightarrow$ Already used in 2-loop $gg \to ZZ$ (talk by Bakul Agarwal), $gg \to ZH$ (2011.12325), $gg \to \gamma\gamma$ (1911.09314), and H + jet (1802.00349).

Adaptive sampling: the problem formulation

Suppose we need to know A_j, some weighted sums of integrals I_i:

$$A_j \equiv \sum_i W_{ji} I_i.$$

Each I_i is a random variable with:

$$I_i \sim \mathscr{N}\left(\mathrm{mean}(I_i)\,, \mathrm{var}(I_i)\right).$$

Assume that $\mathrm{var}(I_i)$ scales with the number of integrand evaluations n_i as

$$\mathrm{var}(I_i) = \frac{w_i}{n_i^\alpha}, \qquad \text{for some } \alpha.$$

Problem: choose n_i (as functions of W_{ji}, w_i, α, and τ_i) to minimize the total integration time T while achieving the requested total variance V_j:

$$T \equiv \sum_i \tau_i n_i \to min, \qquad \mathrm{var}\left(A_j\right) = \sum_i \left|W_{ji}\right|^2 \frac{w_i}{n_i^\alpha} = V_j \ \left(\forall j\right).$$

Adaptive sampling: the solution

Solution: via the Lagrange multiplier method,

$$L \equiv T + \sum_j \lambda_j \left(\mathrm{var}\left(A_j\right) - V_j\right), \qquad \text{and} \qquad \frac{\partial L}{\partial \left\{n_i, \lambda_j\right\}} = 0.$$

Solution if only one A_j is needed:

$$\lambda_j = \frac{1}{\alpha}\left(\frac{1}{V_j}\sum_k \left(\left|W_{jk}\right|^2 w_k \tau_k^\alpha\right)^{\frac{1}{\alpha+1}}\right)^{\frac{\alpha+1}{\alpha}}, \quad n_i = \left(\frac{\alpha w_i}{\tau_i}\lambda_j \left|W_{ji}\right|^2\right)^{\frac{1}{\alpha+1}}.$$

Solution if there are many A_j: only approximate.
In practice pySecDec will:

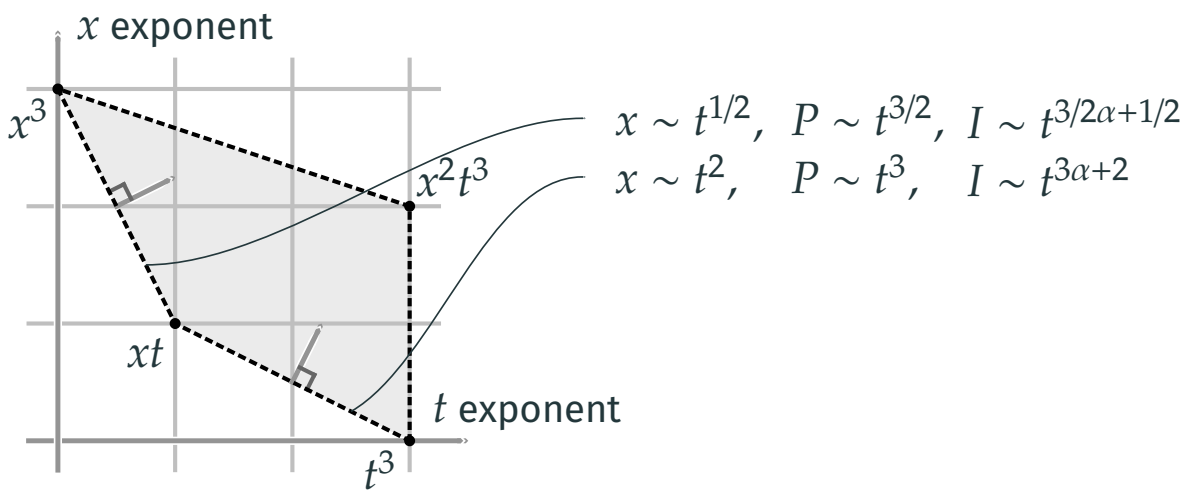
1. Evaluate each I_i on $n_i = 10^4$ points, estimating w_i and τ_i.
2. Get the n_i values as above, but cap them at 20x the previous values.
3. Evaluate I_i using the updated n_i, update w_i and τ_i.
4. Check if $\mathrm{var}\left(A_j\right) \leq V_j$, repeat from 2 if needed.

Asymptotic expansion: finding the regions

Consider an integral depending on *small parameter t* like

$$I = \int_0^1 P^\alpha(x)\,\mathrm{d}x, \qquad P = a\,x^3 + b\,t\,x + c\,t^3 + d\,t^3\,x^2.$$

Plot the *Newton polytope* of P:



From the facets of the polytope we find *two regions*:

1. $x \sim t^{1/2}$, and P can be expanded as $\left(ax^3 + btx\right)\left(1 + \frac{ct^3+dt^3x^2}{ax^3+btx}\right)$;
2. $x \sim t^2$, and P can be expanded as $\left(btx + ct^3\right)\left(1 + \frac{ax^3+dt^3x^2}{btx+ct^3}\right)$.

Note: only valid if $a, b, c > 0$.