Reconstructing complex multi-loop results with FiniteFlow

Tiziano Peraro

HU Berlin/DESY Zeuthen – 23 January 2020

Based on:

T. P., JHEP 1907 (2019) 031, arXiv:1905.08019

Introduction & motivation

Experiments at LHC

- high-accuracy (% level)
- large SM background
- high c.o.m. energy ⇒ multi-particle states

We need scattering amplitudes

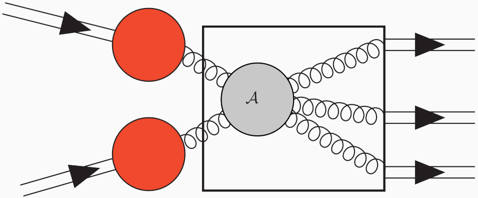
- high accuracy ⇒ loops (% level ∼ 2 loops)
- multi-particle ⇒ high multiplicity

Theoretical studies of amplitudes

- structures of QFT/gauge theories

Scattering amplitudes

- Hadron collider interactions



- Scattering amplitudes
  - main process-dependent part of a physical event
- They can be computed in perturbation theory

$$\mathcal{A} \sim \mathcal{A}_{\text{tree}} + \alpha\, \mathcal{A}_{\text{1-loop}} + \alpha^2\, \mathcal{A}_{\text{2-loops}} + \dots$$

  - %-level accuracy $\sim$ 2 loops

State of the art

- Tree-level and one loop
    - today, mostly numeric
    - essentially solved
    - automated

- Two and higher loops
    - many calculations in recent years . . .
    - . . . but still some open issues
        - until recently, restricted to $2 \to 2$ processes
        - beyond MPLs not well understood

Two and higher loops

- Algebraic calculations for multi-loop amplitudes
    - preferred strategy @ $\ell \geq 2$ loops
        - faster/more stable evaluation
        - better suited for many multi-loop techniques
        - allows more tests, studies, etc. . . and better control
    - often characterized by high complexity

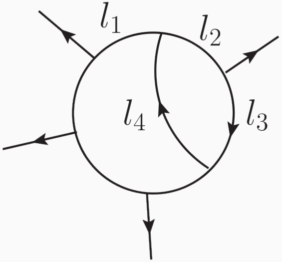
- Complexity can be a combination of
    - number of loops for high accuracy
    - number of legs for high multiplicity
    - numbers of scales (invariants, external/internal masses)

Loop amplitudes

- An integrand contribution to ℓ-loop amplitude

$$\mathcal{A} = \int_{-\infty}^{\infty} \left(\prod_{i=1}^{\ell} d^d k_i \right) \frac{\mathcal{N}}{D_1 \, D_2 \, D_3 \cdots}$$

  - rational function in the components of loop momenta k_j
  - polynomial numerator $\mathcal{N}$
  - quadratic denominators corresp. to loop propagators



$$D_j = l_j^2 - m_j^2$$

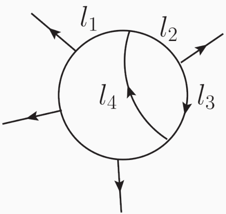
Computing amplitudes: Step 1/3

- Write amplitudes as l.c. of Feynman integrals

$$\mathcal{A} = \sum_j a_j I_j$$

- Dependence on particle-content in rational coeff.s a_j
- The integrals should have a "nice" / "standard" form

$$I = \int_{-\infty}^{\infty} \left(\prod_{i=1}^{\ell} d^d k_i \right) \frac{1}{D_1^{\alpha_1} \, D_2^{\alpha_2} \, D_3^{\alpha_3} \ldots}, \qquad \alpha_j \lesseqgtr 0$$



$$D_j = \begin{cases} l_j^2 - m_j^2 \\ l_j \cdot v_j - m_j^2 \end{cases}$$

Hard to do at
high multiplicity

Computing amplitudes: Step 2/3

Chetyrkin, Tkachov (1981), Laporta (2000)

- Feynman integrals obey linear relations, e.g. IBPs

$$\int \Big(\prod_j d^d k_j\Big) \frac{\partial}{\partial k_j^\mu} v^\mu \frac{1}{D_1^{\alpha_1} D_2^{\alpha_2} \cdots} = 0, \qquad v^\mu = \begin{cases} p_i^\mu & \text{external} \\ k_i^\mu & \text{loop} \end{cases}$$

- Very large and sparse linear systems
- Reduce to linearly independent Master Integrals (MIs) $\{G_1, G_2, \ldots\} \subset \{I_j\}$

$$I_j = \sum_k c_{jk}\, G_k$$

Computing amplitudes: Step 3/3

- The MIs can often be computed analytically
  - in terms of special functions (MPLs, elliptic, ...)
  - most effective method is differential equations (DEs)
    Kotikov (1991), Gehrmann, Remiddi (2000)
  - can be simplified by the choice of MIs, e.g. UT integrals
    Henn (2013)

- Numerical methods may work depending on the process
  - the most successful is sector decomposition
    Binoth, Heinrich (2000)
  - can be improved via IBP reduction to a "better" basis of MIs

Computing amplitudes (summary)

1. Integral representation $\mathcal{A} = \sum_j a_j I_j$
2. IBP reduction $I_j = \sum_k c_{jk} G_k$
3. Compute MIs G_k

A major bottleneck

- Large intermediate expressions
- Intermediate stages much more complicated than final result

Main idea of the talk

- Reconstruct analytic results from "numerical" evaluations
- Can be used for steps 1, 2 and help with step 3 (e.g. using DEs)

Functional reconstruction

- reconstruct analytic results from numerical evaluations
  - evaluation over finite fields $\mathcal{Z}_p$ (i.e. modulo prime integers p)
  - use machine-size integers, $p < 2^{64} \Rightarrow$ fast and exact
  - collect numerical evaluations and infer analytic result
- sidesteps large intermediate expressions & highly parallelizable
- applicable to any rational algorithm
- first applications
  - IBPs and univ. reconstruction von Manteuffel, Schabinger (2014)
  - helicity amplitudes and multivariate reconstruction T.P. (2016)

Some notable examples

- FinRed (private) [von Manteuffel]
  - several results for 4-loop form factors [von Manteuffel, Schabinger]
- FiniteFlow [T.P.]
  - Several two-loop five-point amplitudes [Badger, Brønnum-Hansen, Hartanto, T.P.; Badger, Chicherin, Gehrmann, Heinrich, Henn, T.P., Wasser, Zhang, Zoia]
  - Matter dependence of the four-loop cusp anomalous dimension [Henn, T.P., Stahlhofen, Wasser]
- Private code [Abreu, Dormans, Febres Cordero, Ita, Page, Sotnikov, Zeng]
  - analytic five-parton amplitudes
- Fire 6 [A.V. Smirnov, F.S. Chuharev]
  - Four-loop quark form factor with quartic fundamental colour factor [Lee, Smirnov, Smirnov, Steinhauser]

The black-box interpolation problem

Given a rational function f in the variables $\boldsymbol{z} = (z_1, \ldots, z_n)$ over $\mathcal{Q}$

- Reconstruct analytic form of f, given a numerical procedure

$$(\boldsymbol{z}, p) \longrightarrow \boxed{\quad f \quad} \longrightarrow f(\boldsymbol{z}) \text{ mod } p.$$

- evaluate f numerically for several $\boldsymbol{z}$ and p
- efficient multivariate reconstruction algorithms exist
  e.g. T.P. (2016,2019), Klappert, Lange (2019)
- upgrade analytic f over $\mathcal{Q}$ using rational reconstruction algorithm [Wang (1981)] and Chinese remainder theorem

The black-box interpolation problem

Given a rational function f in the variables $\boldsymbol{z} = (z_1, \ldots, z_n)$ over $\mathcal{Q}$

- Reconstruct analytic form of f, given a numerical procedure

$$(\boldsymbol{z}, p) \longrightarrow \boxed{\quad f \quad} \longrightarrow f(\boldsymbol{z}) \bmod p.$$

- evaluate f numerically for several $\boldsymbol{z}$ and p
- efficient multivariate reconstruction algorithms exist
  e.g. T.P. (2016,2019), Klappert, Lange (2019)
- upgrade analytic f over $\mathcal{Q}$ using rational reconstruction algorithm [Wang (1981)] and Chinese remainder theorem

Question in this talk

How to build the black box?

Example: Scattering amplitudes over finite fields

T.P. (2016)

- External states (momenta and polarizations)
  - rational parametrization with momentum twistors variables
    Hodges (2009), Badger, Frellesvig, Zhang (2013), Badger (2016)
- Tree-level
  - diagrams or recursion relations (e.g. Berends-Giele)
- Loop integrands
  - Feynman diagrams and t'Hooft algebra
  - Unitarity cuts sewing tree-level currents
    - higher finite-dim. representation of internal states in dim. reg.
- Integrand reduction
  - linear fit to a "nice" integrand basis

How to build the black box?

How to build a code for fast numerical evaluations of finite fields?
We can consider a few options:

1. Low-level coding (e.g. in C/C++/FORTRAN)?
   - ✓ very good runtime efficiency
   - ✗ harder to program
   - ✗ limits usability
2. Low-level coding + high-level interfaces?
   - common algorithms in C++ (e.g. linear solvers, fits, etc...)
   - high-level wrapper (e.g. for MATHEMATICA/PYTHON)
   - ✓ good efficiency and usability
   - ✗ not flexible
   - ✗ these algorithms are often intermediate steps

How to build the black box?

Observations:

- A typical multi-loop algorithm involves several steps
  - solving linear systems
  - substitutions / changes of variables
  - etc...
- Large simplifications often occur at the very last stages
  - it's best to do everything numerically
  - only the final expression reconstructed analytically
- Many algorithms share common "building blocks"

FiniteFlow: using data flow graphs

FiniteFlow [T.P. (2019)] has three main components

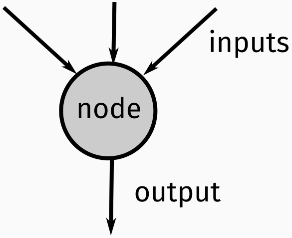
1. "basic" algorithms in C++ over finite fields
    - dense/sparse linear solvers, linear fits, evaluating rat. functions, list manipulations, etc...
2. higher-level framework to combine them into complex ones
    - output of a basic algorithm is input of others
    - graphical representation of your calculation (dataflow graphs)
3. multivariate reconstruction algorithms

FiniteFlow

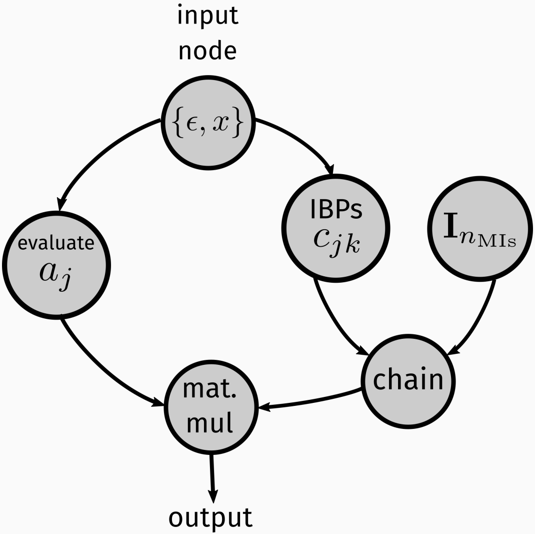
- build complex algorithms without any low-level programming (e.g. from Mathematica interface)
- many methods for amplitudes can be cast in this framework

FiniteFlow: using data flow graphs

- FiniteFlow uses (simplified) data flow graphs
  - Nodes represent numerical algorithms
  - Arrows represent lists of numerical values
- In my implementation, a node has
  - 0 or more lists (arrows) of input values
  - 1 list (arrow) of output values



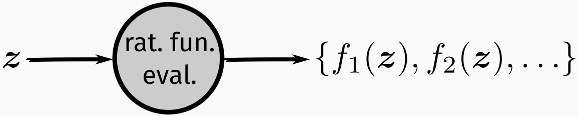
Example of a graph



Example: Evaluation of rational functions

- input: a list of values $\boldsymbol{z} = (z_1, \ldots, z_n)$
- output: a list of rational functions $\{f_1, f_2, \ldots\}$ at $\boldsymbol{z}$

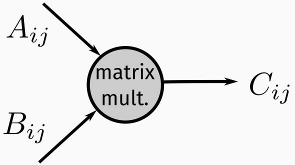
$$f_i(\boldsymbol{z}) = \frac{p_i(\boldsymbol{z})}{q_i(\boldsymbol{z})} = \frac{\sum_\alpha n_{i,\alpha}\, \boldsymbol{z}^\alpha}{\sum_\beta d_{i,\beta}\, \boldsymbol{z}^\beta},$$



Example: Matrix multiplication

- Two lists as input
  1. entries of a matrix A
  2. entries of a matrix B
- use row-major order to store them as a list
- ouput: entries of matrix C such that

$$C_{ij} = \sum_k A_{ik} \, B_{kj}$$



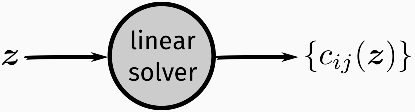
Example: Linear solver

- A $n \times m$ linear system with parametric rational entries

$$\sum_{j=1}^{m} A_{ij}\, x_j = b_i, \quad (i = 1, \ldots, n), \qquad A_{ij} = A_{ij}(\boldsymbol{z}), \quad b_i = b_i(\boldsymbol{z})$$

- input: list of values for paramers $\boldsymbol{z} = (z_1, \ldots, z_n)$
- output: solution $c_{ij} = c_{ij}(\boldsymbol{z})$ such that

$$x_i = \sum_{j \in \text{indep}} c_{ij}\, x_j + c_{i0} \qquad (i \notin \text{indep})$$



Learning algorithms

- Some algorithms have a learning phase
  - used to learn information for defining its output
  - must be completed before using them
- Example: linear solver
  - learn: its rank, dep. and indep. unknowns, indep. eq.s
  - learning phase: solve the system numerically a few times
  - optional: mark & sweep equations (sparse solver)

$\Rightarrow$ It can be used to simplify the algorithm

see also e.g. KIRA: Maierhöfer, Usovitsch, Uwer (2017)

IBP reduction

- IBPs are large and sparse linear systems
- they reduce Feynman integrals I_j to a lin. indep. set of MIs G_j

$$I_i = \sum_j c_{ij}\, G_j$$

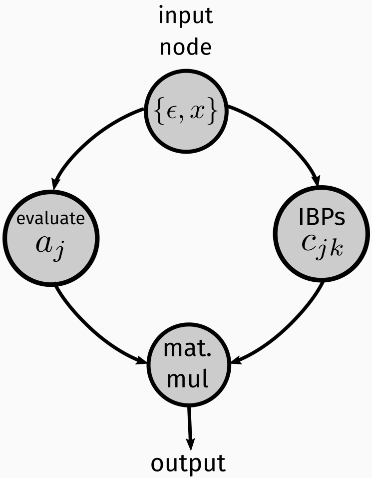
- amplitudes and other multi-loop objects can be reduced mod IBPs

$$A = \sum_j a_j\, I_j = \sum_{jk} a_j\, c_{jk}\, G_k = \sum_j A_j\, G_j\,, \quad \text{with } A_j = \sum_k a_k\, c_{kj}$$

- final results for A_k often much simpler than c_{ij}

$\Rightarrow$ solve IBPs numerically and compute A_j via a matrix multiplication

IBP reduction



Differential equations for MIs

- The MIs G_k satisfy differential equations
  Kotikov (1991), Gehrmann, Remiddi (2000)

$$\partial_x G_i = \sum_j A_{ij}^{(x)} G_j$$

- Identify MIs G_i (e.g. by solving IBPs numerically)
- Compute their derivatives in terms of (non-master) loop integrals

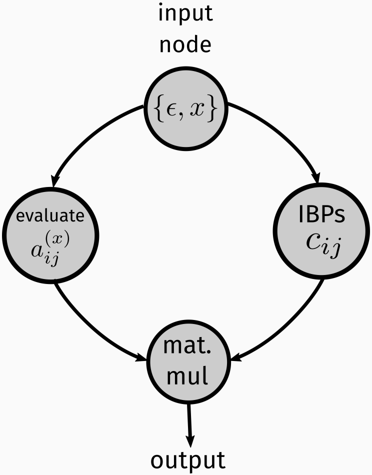
$$\partial_x G_i = \sum_j a_{ij}^{(x)} I_j$$

- Reduce the needed integrals modulo IBPs: $I_i = \sum_j c_{ij} G_j$
- The final result is given by a matrix multiplication

$$A_{ij}^{(x)} = \sum_k a_{ik}^{(x)} c_{kj}$$

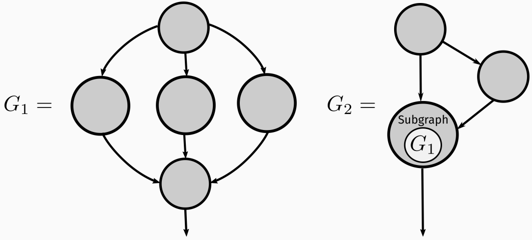
- Reconstruct $A_{ij}^{(x)}$ analytically from its numerical evaluations

Differential equations for MIs



Subgraphs

- Any graph G_1 can be used as a subgraph by an algorithm (a node) A belonging to another graph G_2
  - A will evaluate G_1 several times to compute its output
  - input of G_1 = auxiliary variables chained with inputs of A



Examples:

- Laurent expansion
- maps: evaluate G_1 for several inputs
- partial reconstructions
- (total or partial) fits w.r.t. an ansatz

Coefficients of the ϵ-expansion

- If MIs are known analytically in terms of special functions f_k

$$G_j = \sum_k g_{jk}(\epsilon, x)\, f_k + \mathcal{O}(\epsilon),$$

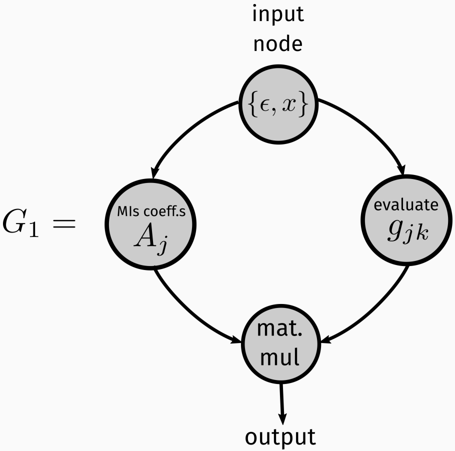
we can compute

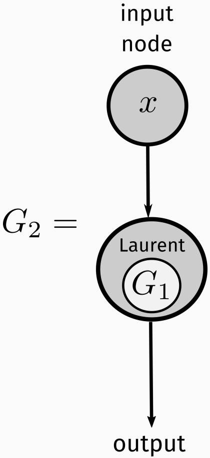
$$A = \sum_k u_k(\epsilon, x)\, f_k + O(\epsilon), \quad \text{where } u_k(\epsilon, x) = \sum_j A_j(\epsilon, x)\, g_{jk}(\epsilon, x)$$

- what we want is the ϵ-expansion of the $u_k(\epsilon, x)$

$$u_k(\epsilon, x) = \sum_{j=-p}^{0} u_k^{(j)}(x)\, \epsilon^j + \mathcal{O}(\epsilon),$$

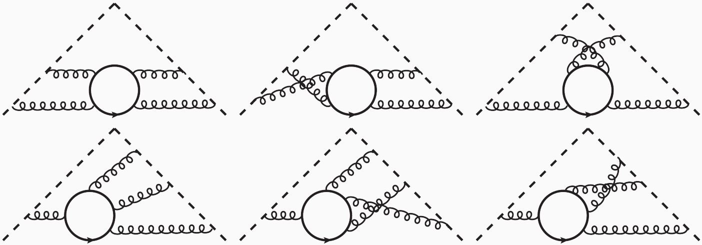
Coefficients of the ϵ-expansion





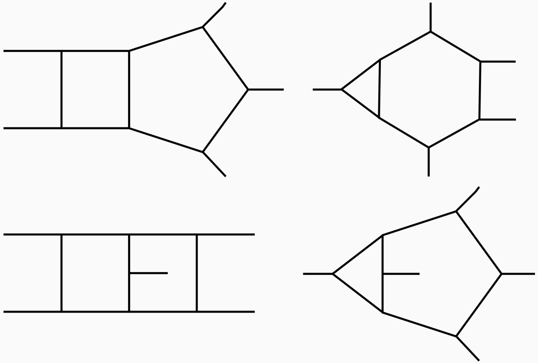
Cutting-edge applications of FiniteFlow

- Matter dependence of the 4-loop cusp anomalous dimension
  [Henn, T.P., Stahlhofen, Wasser (2019)]



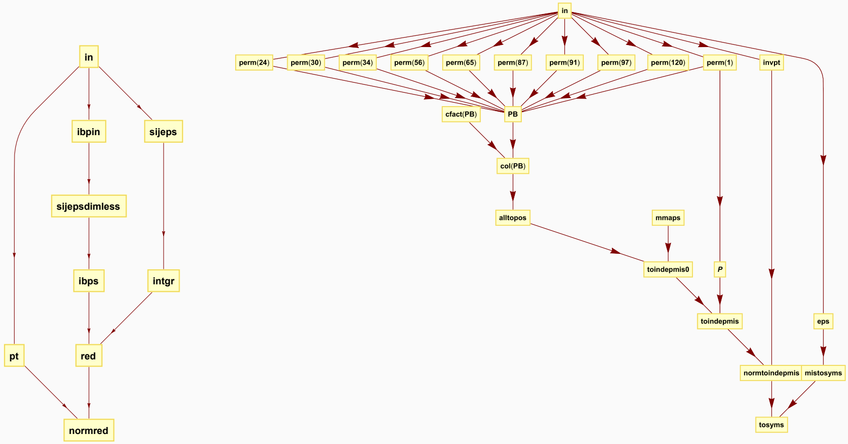
Cutting-edge applications of FiniteFlow

- Five-point two-loop amplitudes
  - Several planar results for five partons and $W + 4$ partons [Badger, Brønnum-Hansen, Hartanto, T.P. (2017-2019)]
  - all-plus five gluon non-planar [Badger, Chicherin, Gehrmann, Heinrich, Henn, T.P., Wasser, Zhang, Zoia (2019)]



Example of graphs in FiniteFlow

Piecing together the all-plus five gluon amplitude (only planar contributions are shown)



Public codes

- FiniteFlow

  https://github.com/peraro/finiteflow

  - C++ code
  - Mathematica interface (strongly recommended)

- FiniteFlow MathTools

  https://github.com/peraro/finiteflow-mathtools

  - packages FFUtils, LiteMomentum, LiteIBP, Symbols
  - examples (amplitudes, IBPs, diff. equations and many more)

Summary & Outlook

Summary

- Finite fields and functional reconstruction
  - enhance the possibilities of our theoretical predictions
  - new results unattainable with traditional computer algebra
  - public code FiniteFlow
- Progress on 2-loop 5-point and other complex processes

Outlook

- More applications
  - massive processes, phase-space integrals, ...
- High level of automation for higher-loop predictions