

Interpolation of Dense and Sparse Rational Functions and other Improvements in **FireFly**

Jonas Klappert, Sven Yannick Klein, and Fabian Lange

*Institute for Theoretical Particle Physics and Cosmology, RWTH
Aachen University, D-52056 Aachen, Germany*

We present the main improvements and new features in version 2.0 of the open-source C++ library **FireFly** for the interpolation of rational functions. This includes algorithmic improvements, e.g. a hybrid algorithm for dense and sparse rational functions and an algorithm to identify and remove univariate factors. The new version is applied to a Feynman-integral reduction to showcase the runtime improvements achieved. Moreover, **FireFly** now supports parallelization with MPI and offers new tools like a parser for expressions or an executable for the insertion of replacement tables.

Contents

1	Introduction	1
2	Algorithmic improvements	2
2.1	Racing Newton vs Ben-Or/Tiwari	2
2.1.1	Ben-Or/Tiwari algorithm	3
2.1.2	Berlekamp-Massey algorithm	5
2.1.3	Early termination in the Ben-Or/Tiwari algorithm	6
2.1.4	Racing	6
2.1.5	Comparison of Newton and racing algorithm	7
2.2	Interpolation of sparse and dense multivariate rational functions	8
2.3	Finding factors	10
3	Technical improvements and new features	14
3.1	New interface and overhead reduction	14
3.2	Parsing collections of rational functions	16
3.3	MPI	19
4	Applications in IBP reductions	21
4.1	IBP reduction with Kira and FireFly	23
4.2	Insertion of IBP tables	25
5	Conclusions	27
A	Mathematica code to generate dense polynomials	29

1 Introduction

The general interpolation of unknown polynomials based on their evaluations, i.e. *probes*, dates back to Newton’s research in the 17th century and the algorithm named after him [1]. This problem class is known as *black box* interpolation. In the last fifty years, several sophisticated interpolation algorithms for multivariate polynomials have been developed, improved, and combined, e.g. in Refs. [2–8]. Some of the algorithms are designed for *dense* polynomials where most of the monomials up to the degree of the polynomial are nonzero, some are designed for *sparse* polynomials where only few monomials are nonzero, and some for both. In contrast, the interpolation of rational functions is a younger field of research dating back to Thiele’s formula from the beginning of the 20th century [1]. The first algorithms for multivariate rational functions were developed thirty years ago [9–12] and since then more algorithms have been published [13–16]. Most of them rely on one or more polynomial-interpolation algorithms as sub-algorithms. Again, some of the algorithms are designed for sparse and others for dense functions. Moreover, most of the algorithms mentioned above are designed with finite-field arithmetic in mind, e.g. integers modulo a prime number. The elements of a finite field can be promoted to rational numbers with rational reconstruction algorithms [17, 18].

Last year, two of the authors of the present paper published the C++ library **FireFly** for the interpolation of rational functions [19] based on the sparse rational-function-interpolation algorithm by Cuyt and Lee [15] with an embedded Zippel algorithm for the polynomials [2, 3]. Shortly after, the C++ library **FiniteFlow** was published [20]. Even though it uses the central idea of Ref. [15], it is a dense implementation.

In this paper, we present version 2.0 of **FireFly** which includes both algorithmic and technical improvements. The algorithmic improvements are presented in Sect. 2. First, we embedded the univariate racing algorithm of Kaltofen, Lee, and Lobo [6, 7] into the multivariate Zippel algorithm. It races the dense Newton algorithm against the sparse Ben-Or/Tiwari algorithm [4]. This significantly improves the performance of **FireFly** for sparse functions. Secondly, we present a modified version of the algorithm of Ref. [15] based on the *temporary pruning* idea of Refs. [6, 7]. Instead of only pruning on the level of polynomials, we now also prune on the level of rational functions. This increases the performance especially for dense functions. Lastly, we implemented an algorithm to search for univariate factors of the rational function before the actual multivariate interpolation. By removing those factors, the interpolation can become much simpler. In the context of integration-by-parts (IBP) reductions in theoretical particle physics [21, 22], similar ideas have been pursued in Refs. [23, 24].

In Sect. 3 we summarize the main technical improvements since the release of **FireFly**. First, we changed the interface to pave the way for a reduction of overheads inspired by the vectorization of modern CPUs. Instead of evaluating the black box at one parameter point, one can evaluate it at several points at the same time. This reduces the overhead of many problems, most notably when solving system of equations. Secondly, we implemented a parser which allows the user to read in expressions from files and strings and use them as building block for the black box. The parser is designed for fast evaluations. Lastly, **FireFly** now supports the parallelization with MPI. This enables **FireFly** to utilize computer clusters

for the black-box evaluations. Since they are the main bottleneck of interpolation problems in a physical context, this can significantly speed up calculations.

In precision high-energy physics, integral reductions based on IBP relations [21, 22] play a major role. The most prominent strategy is the Laporta algorithm [25], which is implemented in several public [26–34] and numerous private codes. In the last decade, finite-field and interpolation techniques spread to this field beginning with Ref. [35]. The former can be used to remove redundant information to reduce the size of the system of equations [36] as implemented in *Kira* [33, 34]. By directly interpolating the final result, one can also circumvent the large intermediate expressions which usually appear [37, 38]. *FIRE6* is the first public tool available with interpolation techniques implemented for IBP reductions [30]. These techniques also facilitate the development of new strategies for IBP reductions, see e.g. Refs. [39, 40]. Another bottleneck can be the insertion of the final reduction tables into expressions like amplitudes. Also, this problem might be eased with the help of finite-field arithmetic and interpolation techniques. In Sect. 4 we apply the current development version of *Kira* to a reduction problem to illustrate the improved performance of *FireFly* 2.0 and describe a tool based on *FireFly* 2.0, which can insert replacement tables into expressions.

Recently, a similar strategy to obtain amplitudes from floating-point probes has been presented [41, 42].

2 Algorithmic improvements

In the following sections we describe the algorithmic improvements that have been implemented since the release of the first version of *FireFly*. We refer the reader to Ref. [19] and the references therein for a description of the base algorithms implemented in *FireFly* and our notation. The interpolation algorithms presented here are discussed over a prime field \mathbb{Z}_p with characteristic p , where p is its defining prime.

2.1 Racing Newton vs Ben-Or/Tiwari

As sub-algorithm for the interpolation of rational functions with *FireFly* we employ a multivariate polynomial interpolation with the Zippel algorithm [2, 3]. The Zippel algorithm itself requires a univariate polynomial-interpolation algorithm. Instead of a univariate interpolation with the Newton algorithm, we now employ the racing algorithm presented in Refs. [6, 7]. It races the dense Newton interpolation against the Ben-Or/Tiwari algorithm [4]. The Ben-Or/Tiwari algorithm is a sparse polynomial-interpolation algorithm and scales with the number of terms, whereas the Newton interpolation scales with the degree of the polynomial. The racing algorithm updates both interpolation algorithms sequentially with a new probe until either of them terminates. Hence, it performs well for both sparse and dense polynomials.

In the following sections, we summarize both the univariate Ben-Or/Tiwari algorithm and its sub-algorithms based on Ref. [5, 6] and the racing algorithm of Refs. [6, 7].

2.1.1 Ben-Or/Tiwari algorithm

The Ben-Or/Tiwari algorithm is an interpolation algorithm for multivariate Polynomials [4]. However, for our purpose the univariate case is sufficient. Consider the polynomial

$$f(z) = \sum_{i=1}^T c_{\alpha_i} z^{\alpha_i} \quad (1)$$

with T nonzero terms. The c_{α_i} are the coefficients of the monomials of degrees α_i . For an arbitrary element $y \in \mathbb{Z}_p$, the anchor point, we then define the evaluations of the polynomial at integer powers $i \geq 1$ of y as

$$a_i = f(y^i) . \quad (2)$$

The original formulation of the algorithm uses $i \geq 0$. However, then it is more vulnerable to accidental cancellations, because the first evaluation is performed at $y^0 = 1 \forall y$ [43].

The main ingredient of the Ben-Or/Tiwari algorithm is the auxiliary polynomial

$$\zeta(z) = \prod_{i=1}^T (z - y^{\alpha_i}) = z^T + \lambda_{T-1} z^{T-1} + \dots + \lambda_0 , \quad (3)$$

with the coefficients λ_k . It is designed such that one can obtain the degrees α_i of the polynomial $f(z)$ by computing the roots of $\zeta(z)$. Hence, by definition,

$$\sum_{i=0}^T c_{\alpha_i} (y^{\alpha_i})^j \zeta(y^{\alpha_i}) = 0 \quad (4)$$

holds for any integer $j \geq 0$. On the other hand, this sum can be rewritten as

$$\begin{aligned} \sum_{i=0}^T c_{\alpha_i} (y^{\alpha_i})^j \zeta(y^{\alpha_i}) &= \sum_{k=0}^{T-1} \lambda_k \left(\sum_{i=0}^T c_{\alpha_i} (y^{\alpha_i})^{j+k} \right) + \sum_{i=0}^T c_{\alpha_i} (y^{\alpha_i})^{j+T} \\ &= \sum_{k=0}^{T-1} \lambda_k \left(f(y^{j+k}) \right) + f(y^{j+T}) , \end{aligned} \quad (5)$$

with the definition of $f(z)$ in Eq. (1). Therefore, one obtains the linear relation

$$0 = \sum_{k=0}^{T-1} \lambda_k a_{j+k} + a_{j+T} \quad (6)$$

between the coefficients λ_k of the auxiliary polynomial. This allows for the construction of a system of equations based on the evaluations a_{j+k} of the polynomial $f(z)$. Since there are exactly T unknowns, T linearly independent equations of this form are required. The minimum number of evaluations can be achieved by constructing one equation with $T + 1$ evaluations a_{j+k} and reusing T of them for the next equation. Thus, one only has to compute one additional probe for each equation after the first. Therefore, in total $2T$

probes are required to find the auxiliary polynomial $\zeta(z)$. The system of T equations can then be written in matrix form as

$$\begin{pmatrix} a_1 & a_2 & \cdots & a_T \\ a_2 & a_3 & \cdots & a_{T+1} \\ \vdots & \vdots & \ddots & \vdots \\ a_T & a_{T+1} & \cdots & a_{2T-1} \end{pmatrix} \begin{pmatrix} \lambda_0 \\ \lambda_1 \\ \vdots \\ \lambda_{T-1} \end{pmatrix} = - \begin{pmatrix} a_{T+1} \\ a_{T+2} \\ \vdots \\ a_{2T} \end{pmatrix}. \quad (7)$$

Eq. (7) is a Hankel system which can be solved iteratively with the Berlekamp-Massey algorithm (Alg. (1)) described in the next section. This can be understood by the fact that the λ_k of Eq. (6) build a linear-feedback shift register that generates the sequence a_i .

Once the auxiliary polynomial of Eq. (3) has been calculated, its roots have to be obtained. Since the roots are integer powers i of y by construction, the simplest method is to evaluate $\zeta(z)$ at these values and check whether the result is zero. Hence, one starts with $y^{i=0} = 1$ and increases the power i until the number of roots matches the degree of $\zeta(z)$. The powers i corresponding to the roots are the degrees α_i of the monomials of $f(z)$ and the number of roots corresponds to the number of terms T .

The coefficients c_{α_i} of $f(z)$ can be obtained by constructing and solving a system of the equations

$$a_i = f(y^i) = \sum_{j=1}^T c_{\alpha_j} (y^i)^{\alpha_j}. \quad (8)$$

By choosing a sequence of i , one can construct a shifted transposed Vandermonde system for the coefficients c_{α_i} [43]. We choose the probes $1 \leq i \leq T$ in our implementation. A solving algorithm for these systems has already been implemented in the first version of **FireFly** [19]. No additional probes are required for this step because all previously obtained probes can be reused.

The case $f(z) = 0$ is special because the Hankel matrix in Eq. (7) becomes a null matrix. Therefore, $\zeta(z)$ is undetermined. However, the Berlekamp-Massey algorithm yields $\zeta(z) = 1$, which has no roots. Since this is the only case without roots, one can easily intercept it.

Since the interpolation is performed over finite fields, one has to account for the fact that the element y may generate a cyclic group with a small order, i.e. there is a small power b for which $y^b = 1$. Hence, if y^j is a root of the auxiliary polynomial also $y^{(j+n \cdot b)} \forall n \in \mathbb{N}$ are roots of the auxiliary polynomial, which is problematic if b is smaller than the degree of the polynomial. Additionally, if the power b is smaller than the number of terms T , the equations in the Hankel system of Eq. (7) and the Vandermonde system are not linearly independent. Therefore, it is necessary to use fields with a large characteristic so that the integer b is larger than the degree and the number of terms of the interpolated polynomial for the anchor point y with high probability [7].

The Ben-Or/Tiwari algorithm can be illustrated as follows. Assume the polynomial

$$f(z) = z^2 + 1 \quad (9)$$

defined over the field \mathbb{Z}_{509} shall be interpolated. We choose $y = 2$ as the anchor point. Since there are $T = 2$ terms, $2T = 4$ probes are required:

$$f(y^1) = 5, \quad f(y^2) = 17, \quad f(y^3) = 65, \quad f(y^4) = 257. \quad (10)$$

With this input, the Berlekamp-Massey algorithm leads us to

$$\zeta(z) = z^2 + 504z + 4 \quad (11)$$

as the auxiliary polynomial. Evaluating $\zeta(z)$ at integer powers of y until two roots are found yields

$$\zeta(y^0) = 0, \quad \zeta(y^1) = 507, \quad \zeta(y^2) = 0. \quad (12)$$

Thus, the polynomial consists of monomials with degrees 0 and 2. Their coefficients are then obtained by solving the system

$$f(2^1) = 5 = c_0 + c_2 (2^1)^2, \quad f(2^2) = 17 = c_0 + c_2 (2^2)^2, \quad (13)$$

which yields $c_0 = c_2 = 1$.

2.1.2 Berlekamp-Massey algorithm

The Berlekamp-Massey algorithm was originally a decoding algorithm in coding theory designed by Berlekamp [44]. Massey simplified and connected it to linear-feedback shift registers [45]. Later, it was interpreted in terms of Hankel systems [46, 47].

The Berlekamp-Massey algorithm is an iterative algorithm to find the linear generator

$$\Lambda(z) = \lambda_0 z^T + \dots + \lambda_{T-1} z + \lambda_T \quad (14)$$

of a stream of elements a_1, a_2, \dots given by Eq. (6) over an arbitrary field. Although the sequence is unbound, the algorithm will compute the polynomial $\Lambda(z)$ after processing $2T$ elements from the input stream. The degree of $\Lambda(z)$ is the length L of the linear-feedback shift register. While processing a new a_r as a new iteration step, Eq. (6) is checked, i.e. the r.h.s. of Eq. (6) is evaluated using the polynomial $\Lambda_{r-1}(z)$ obtained in the previous iteration step of the algorithm. This yields the discrepancy

$$\Delta_r \equiv \sum_{k=0}^{L-1} \lambda_k a_{j+k} + a_{j+L}, \quad (15)$$

with the λ_k being the coefficients of the current polynomial $\Lambda_{r-1}(z)$ and $j = r - L$. Note that $\lambda_L = 1$ always holds. If Δ_r vanishes, the polynomial $\Lambda_{r-1}(z)$ is a valid generator for the sequence of a_i up to a_r and does not need to be changed. If $\Delta_r \neq 0$, there are two update possibilities depending of the current length L_{r-1} . The first update increases the generator's degree and the second update adjusts the lower orders of the generator's coefficients. The details of the update steps are summarized in Alg. (1).

Thus, the Berlekamp-Massey algorithm determines the coefficients λ_k in Eq. (6), which is equivalent to solving the Hankel system given by Eq. (7) [46, 47]. The auxiliary polynomial $\zeta(z)$ of the Ben-Or/Tiwari algorithm can be obtained by reversing the coefficients in $\Lambda(z)$, i.e.

$$\zeta(z) = \lambda_T z^T + \dots + \lambda_1 z + \lambda_0 \quad (16)$$

with $\lambda_T \equiv 1$.

Algorithm 1 Berlekamp-Massey algorithm from Ref. [6].

Input: a_1, a_2, \dots

Initialization: $\Lambda_0(z) \leftarrow 1; B_0(z) \leftarrow 0; L_0 \leftarrow 0; \Delta \leftarrow 1;$

for $r = 1, 2, \dots$ **do**

(Calculate the discrepancy Δ_r , assuming that $\Lambda_{r-1}(z) = \lambda_s + \lambda_{s-1}z + \dots + \lambda_0z^s$)

$\Delta_r \leftarrow \lambda_s a_r + \lambda_{s-1} a_{r-1} + \dots + \lambda_0 a_{r-s};$

if $\Delta_r = 0$ **then**

$\Lambda_r(z) \leftarrow \Lambda_{r-1}(z); B_r(z) \leftarrow zB_{r-1}(z); L_r \leftarrow L_{r-1};$

else if $\Delta_r \neq 0$ and $2L_{r-1} < r$ **then**

$B_r(z) \leftarrow \Lambda_{r-1}(z); \Lambda_r(z) \leftarrow \Lambda_{r-1}(z) - \frac{\Delta_r}{\Delta} zB_{r-1}(z); L_r \leftarrow r - L_{r-1}; \Delta \leftarrow \Delta_r;$

else if $\Delta_r \neq 0$ and $2L_{r-1} \geq r$ **then**

$\Lambda_r(z) \leftarrow \Lambda_{r-1} - \frac{\Delta_r}{\Delta} zB_{r-1}(z); B_r \leftarrow zB_{r-1}(z); L_r \leftarrow L_{r-1};$

end if

end for

2.1.3 Early termination in the Ben-Or/Tiwari algorithm

The Ben-Or/Tiwari algorithm described in Sect. 2.1.1 requires the number of terms T as input parameter. However, in Refs. [6, 7] it was formulated in an *early termination* version, which does not rely on T as input.

The first step of the Ben-Or/Tiwari interpolation is the Berlekamp-Massey algorithm to compute the auxiliary polynomial $\zeta(z)$. Alg. (1) processes one probe after another and does not rely on subsequent probes. If the discrepancy $\Delta_r = 0$, $\Lambda_{r-1}(z)$ already solves the Hankel system of Eq. (7). This happens either accidentally or if the number of probes reached $2T + 1$. The probability of the former is less than [6, 7]

$$\frac{T(T+1)(2T+1)D}{6 \cdot \#\mathbb{F}}, \quad (17)$$

where D is the degree of the polynomial and $\#\mathbb{F}$ is the characteristic of the field \mathbb{F} .

Thus, the termination criterion is to terminate after a sequence of η probes yields $\Delta_r = 0$ and the total number of probes with non-vanishing discrepancy corresponds to a Hankel system, i.e. $r = 2L + \eta$ with $L \in \mathbb{N}$. In our experience, it is sufficient to choose $\eta = 1$, when interpolating over fields with 63-bit characteristics. With the auxiliary polynomial $\zeta(z)$ known, the Ben-Or/Tiwari algorithm can proceed. By definition, $\zeta(z)$ has to completely factorize into mutually distinct factors $z - y^i$. There is an upper bound on the power i if there is an upper bound on the degree of the polynomial, which is the case when embedding the Ben-Or/Tiwari interpolation into the interpolation of a rational function following the strategy of Ref. [15] (see Sect. 2.2 for details). If the auxiliary polynomial completely factorizes in this way, the early termination was correct with high probability. If the factorization fails, we take the result of the Newton interpolation which always terminates.

2.1.4 Racing

As proposed in Ref. [6, 7], we race the Newton and the Ben-Or/Tiwari algorithm against each other. Our implementation of the former and its early termination criterion are described

in Ref. [19]. For each probe, both algorithms get updated with this numerical evaluation. If either of them reaches its termination criterion, we abort the other interpolation and take the result of the terminated algorithm.

The advantage of the racing algorithm is that it minimizes the number of probes required to interpolate a polynomial. The Newton interpolation is a dense algorithm and takes $D + 1 + \eta$ probes with the termination criterion, i.e. one probe for the term of each degree up to the maximum degree D plus η additional checks. On the other hand, the Ben-Or/Tiwari interpolation finishes after $2T + \eta$ probes as described in the previous section and terminates much earlier than Newton’s interpolation if $T \ll D$. Therefore, the racing algorithm scales well for both dense and sparse polynomials. Moreover, when embedding the racing algorithm into the multivariate Zippel algorithm, it usually scales better than both algorithms alone, because a different algorithm can win for different variables [6, 7].

This improvement comes with additional runtime for the Ben-Or/Tiwari algorithm as the only drawback. However, this additional runtime is regained manifold if the Ben-Or/Tiwari interpolation finishes faster than Newton’s interpolation sometimes and the runtime for each probe dominates the total runtime of the interpolation.

2.1.5 Comparison of Newton and racing algorithm

To illustrate the changes in the number of required black-box probes, we compare **FireFly** 1.1 with and without the racing algorithm. When no racing algorithm is used, the Newton interpolation is used by default. We test the benchmark functions

$$f_1(z_1, \dots, z_{20}) = \frac{\sum_{i=1}^{20} z_i^{20}}{\sum_{i=1}^{10} z_i^{20} - \sum_{i=11}^{20} z_i^{20}}, \quad (18)$$

$$f_2(z_1, \dots, z_5) = \frac{z_1^{100} + z_2^{200} + z_3^{300}}{z_1 z_2 z_3 z_4 z_5 + z_1^4 z_2^4 z_3^4 z_4^4 z_5^4}, \quad (19)$$

$$f_3(z_1, \dots, z_5) = \frac{(1 + z_1 + z_2 + z_3 + z_4 + z_5)^{17} - 1}{z_4 - z_2 + z_1^{10} z_2^{10} z_3^{10} z_4^{10} z_5^{10}} \quad (20)$$

and enable the scan for a sparse shift by default. The results are summarized in Tab. 1.

Table 1: Comparison of Newton and racing in **FireFly** 1.1 with respect to runtime and required probes for the benchmark functions defined by Eqs. (18)–(20). The interpolation is performed over $\mathbb{Z}_{9223372036854775783}$ using a single thread of an AMD Ryzen 5 3400G processor.

Function	Newton		Racing	
	Probes	Runtime	Probes	Runtime
f_1	15242	0.33 s	3082	0.15 s
f_2	134906	30 s	1957	0.6 s
f_3	102358	20 s	101926	20 s

Since the Newton interpolation is a dense interpolation, it requires up to two orders of magnitude more probes compared to the racing algorithm for sparse functions like f_1 and

f_2 . Consequently, the racing algorithm can improve the runtime significantly in such cases. For dense functions like f_3 , there is almost no difference between the two algorithms in the number of required black-box probes. Therefore, the additional runtime for the Ben-Or/Tiwari algorithm is negligible in most applications, especially if the computation of the probes is dominant compared to the runtime of the interpolation.

2.2 Interpolation of sparse and dense multivariate rational functions

In this section, we present a modified version of the interpolation algorithm developed by Cuyt and Lee [15] that improves the scaling behavior for dense rational functions. Our new algorithm is designed to find a balance between the required number of black-box probes for dense and sparse rational functions.

In practice, it is based on the *temporary pruning* idea of Refs. [6, 7], i.e. the coefficients which have already been interpolated are removed from the system of equations. Consequently, we race a dense interpolation, i.e. including the effects of the variable shift, against the sparse approach by Ref. [15]. The algorithm can be divided into the following steps:

1. Find a variable shift $\vec{s} = (s_1, \dots, s_n)$ for a given black-box function $f(z_1, \dots, z_n)$ of n variables, that generates a constant term in either numerator or denominator and involves a minimal set of nonzero $s_i, i \in [1, n]$.
2. Interpolate the univariate auxiliary function

$$\tilde{f}(t\vec{z} + \vec{s}) = f(tz_1 + s_1, \dots, tz_n + s_n) \quad (21)$$

in the homogenization variable t [48] at numerical values for \vec{z} . One can set $z_1 = 1$ and restore its dependence later by homogenization. The interpolation of \tilde{f} is only performed once and then replaced by solving a system of equations built by all nonzero degrees in t that still need to be processed by multivariate polynomial interpolations.

3. Process all coefficients of \tilde{f} by multivariate polynomial interpolations simultaneously including the effects of the shift. Hence, all but the highest degree terms in numerator and denominator are interpolated densely.
4. When an interpolation terminates, restore its dependence in z_1 by homogenization. When the interpolation of the polynomial coefficient with the currently highest degree of \tilde{f} , labeled f_h , in either numerator or denominator terminates, abort the interpolation of the polynomial coefficient of the next-to-highest degree, labeled f_l . Calculate the monomials that occur by shifting f_h and restart the interpolation of f_l while subtracting the shift monomials of all higher degree terms numerically. Hence, this is now a sparse interpolation. The probes processed during the dense interpolation can be reused. If the stored probes are not sufficient for the restarted interpolation to terminate, proceed.
5. Prune the system of equations for the auxiliary function \tilde{f} by all interpolated polynomial coefficients. When a coefficient has been interpolated sparsely, i.e. the monomials created by shifts in higher degree polynomials were subtracted, re-add those monomials to the system of equations for consistency. While pruning the system of equations,

the already solved polynomials created by the shift have to be evaluated at $t\vec{z}$ instead of $t\vec{z} + \vec{s}$.

6. After all interpolations terminated, subtract all polynomials generated by shifts of higher degree coefficients from the coefficients which have been interpolated densely.

This algorithm is applicable for both dense and sparse rational functions and avoids over-sampling for dense functions, which is apparent in the algorithm of Ref. [15], while still maintaining a good performance for sparse functions. For further reference, we call this algorithm *hybrid racer*, because it races sparse and dense interpolations while being efficient for both cases.

In our implementation, we use the multivariate Zippel algorithm as described in Ref. [19] with the embedded univariate racing algorithm described in Sect. 2.1 for the polynomial interpolation in point 3. Since we interpolate the functional dependence of each variable sequentially ($z_2 \rightarrow z_3 \rightarrow \dots \rightarrow z_n$) and the polynomial coefficients proceed differently in general, we have to choose for which coefficient we compute the next probes. We prefer to schedule probes for z_i , where i is the index closest to n . This ensures that the total number of probes is bounded by the maximum number of possible terms of a dense rational function with given degrees.

To illustrate this algorithm, consider the black-box function

$$f(z_1, z_2) = \frac{z_1 z_2^3 + z_1^2 z_2^2 + z_1^3 z_2 + z_1^4 + z_2^5}{z_2}. \quad (22)$$

To fix the normalization, we shift z_2 as $z_2 + 2$ yielding

$$f(z_1, z_2 + 2) = \frac{z_1(z_2 + 2)^3 + z_1^2(z_2 + 2)^2 + z_1^3(z_2 + 2) + z_1^4 + (z_2 + 2)^5}{2 + z_2}. \quad (23)$$

After performing one interpolation of Eq. (23) as a function of t with eleven probes, one obtains the skeleton of the auxiliary rational function

$$\tilde{f}(t, tz_2 + 2) = \frac{n_{\alpha_0} + n_{\alpha_1}t + n_{\alpha_2}t^2 + n_{\alpha_3}t^3 + n_{\alpha_4}t^4 + n_{\alpha_5}t^5}{1 + d_{\beta_1}t}, \quad (24)$$

where each n_α and d_β is a multivariate polynomial, i.e. $n_\alpha \equiv n_\alpha(z_1, z_2)$. Hence, we can build a system of equations, where each equation is structured as

$$n_{\alpha_{1,u}}t + n_{\alpha_{2,u}}t^2 + n_{\alpha_{3,u}}t^3 + n_{\alpha_{4,u}}t^4 + n_{\alpha_{5,u}}t^5 - d_{\beta_{1,u}}t f(t, tz_2 + 2) = -n_{\alpha_{0,s}} + f(t, tz_2 + 2). \quad (25)$$

The index s (u) indicates a solved (unsolved) coefficient. After the interpolation in t , the constant in the numerator, $n_{\alpha_{0,s}}$, is already determined and can be used to prune the system by one term. Thus we need only six black-box probes to solve the system of equations defined by Eq. (25), whereas the approach by Ref. [15] requires seven. After solving one system built by Eq. (25), the linear terms n_{α_1} and d_{α_1} terminate their interpolations by reaching the degree bound in Newton's algorithm. Therefore, the system can be pruned to size four:

$$n_{\alpha_{2,u}}t^2 + n_{\alpha_{3,u}}t^3 + n_{\alpha_{4,u}}t^4 + n_{\alpha_{5,u}}t^5 = -n_{\alpha_{0,s}} - n_{\alpha_{1,s}}t + (1 + d_{\beta_{1,s}}t)f(t, tz_2 + 2). \quad (26)$$

After probing the black box four additional times, the system built by Eq. (26) can be solved. With these new probes, the Ben-Or/Tiwari algorithm for n_{α_5} terminates and the Newton algorithm for n_{α_2} reaches its degree bound. Thus, the system can be pruned again and reduces to

$$n_{\alpha_{3,u}}t^3 + n_{\alpha_{4,u}}t^4 = -n_{\alpha_{0,s}} - n_{\alpha_{1,s}}t - n_{\alpha_{2,s}}t^2 - n_{\alpha_{5,s}}t^5 + (1 + d_{\beta_{1,s}}t)f(t, tz_2 + 2) \quad (27)$$

with two unsolved coefficients. Since the highest degree polynomial of the numerator, $n_{\alpha_{5,s}}$, is now known, we can abort the interpolation of n_{α_4} and restart it with the previously computed values, while numerically subtracting the effects of shifting z_2 in $n_{\alpha_{5,s}}$. After the next solution of Eq. (27), n_{α_3} terminates by reaching the degree bound. Thus, the system can be pruned to

$$n_{\alpha_{4,u}}t^4 = -n_{\alpha_{0,s}} - n_{\alpha_{1,s}}t - n_{\alpha_{2,s}}t^2 - n_{\alpha_{3,s}}t^3 - n_{\alpha_{5,s}}t^5 + (1 + d_{\beta_{1,s}}t)f(t, tz_2 + 2). \quad (28)$$

Solving Eq. (28), allows the Newton interpolation of n_{α_4} to finish with the early termination criterion. Afterwards, one has to calculate the shift expansion to remove its effects on all coefficients except n_{α_4} , n_{α_5} , and d_{β_1} .

In total, the hybrid racer algorithm requires 24 probes to interpolate the function given by Eq. (22). Note that shifting both z_1 and z_2 does not increase the number of required probes. In contrast, our implementation of the algorithm presented in Ref. [15] in **FireFly 1.1** requires 39 black-box probes in total using the same variable shift. There, the interpolation of \tilde{f} with Thiele's formula is replaced by a system of equations with seven unknowns after the first interpolation. Depending on the sparse shift, the hybrid racer algorithm could potentially prefer to interpolate a rational function densely although the sparse approach by Ref. [15] would lead to fewer black-box probes. However, the imposed oversampling should still reflect the sparsity of the rational function.

To illustrate the performance of the hybrid-racer algorithm implemented in **FireFly 2.0** compared to **FireFly 1.1**, Tab. 2 shows a comparison in runtime and required black-box probes for the benchmark functions defined in Eqs. (18)–(20). In addition, we include the function

$$f_4(z_1, \dots, z_5) = \frac{(1 + z_1 + z_2 + z_3 + z_4 + z_5)^{20} - 1}{z_4 - z_2 + z_1^{10} z_2^{10} z_3^{10} z_4^{10} z_5^{10}}. \quad (29)$$

to exemplify the scaling w.r.t. Eq. (20) by doubling the number of monomials. Apart from runtime improvements during the development of **FireFly 2.0**, the reduced amount of probes for dense functions is evident. Since the hybrid-racer algorithm is designed to be bounded at the maximum number of possible monomials, for certain types of sparse functions, the number of required black-box probes can increase in comparison to the algorithm of Ref. [15]. However, some of these drawbacks can be circumvented when combining it with the scan for factors described in Sect. 2.3.

2.3 Finding factors

The rational functions that occur in physical calculations usually include polynomial factors in the given set of variables. Once these factors are found, they lead to simpler results

Table 2: Comparison of runtime and number of required probes for the benchmark functions defined by Eqs. (18)–(20) and Eq. (29) between FireFly 1.1 and FireFly 2.0. The interpolation is performed over $\mathbb{Z}_{9223372036854775783}$ using a single thread of an AMD Ryzen 5 3400G processor.

Function	FireFly 1.1		FireFly 2.0	
	Probes	Runtime	Probes	Runtime
f_1	3082	0.2 s	951	0.02 s
f_2	1957	0.5 s	1285	0.3 s
f_3	101926	20 s	26739	7.9 s
f_4	212698	79 s	53213	28 s

and can be removed to reduce the number of probes required by black-box interpolations. Recently, Refs. [23, 24] have shown that in the context of IBP reductions denominators of rational functions can factorize completely by choosing a specific basis of master integrals.

In this section, we describe an algorithm implemented in FireFly that searches for univariate polynomial factors of black-box functions automatically as an additional scan before the actual interpolation. These factors are then canceled in the interpolation to reduce the number of black-box probes and to simplify the results. Factors including two or more variables are not covered by our algorithm. This restriction allows us to utilize the FLINT library [49, 50] for polynomial factorizations over finite fields. The factors in z_i are calculated as follows:

1. Replace the variable set $\{z_1, \dots, z_n\} \setminus z_i$ by distinct random numbers $\{r_{j,1}\} \in \mathbb{Z}_{p_1}$.
2. Evaluate the black box with distinct, randomized values t at $\vec{z} = \{r_{1,1}, \dots, r_{i-1,1}, t, r_{i+1,1}, \dots, r_{n,1}\}$ and perform a univariate rational-function interpolation to obtain a rational function in z_i labeled as $f(z_i; \{r_{j,1}\}; p_1)$.
3. Calculate the factors of numerator and denominator of $f(z_i; \{r_{j,1}\}; p_1)$ in z_i .
4. Generate a new set of random numbers $\{r_{j,2}\} \in \mathbb{Z}_{p_1}$ and perform a second interpolation to obtain the rational function $f(z_i; \{r_{j,2}\}; p_1)$. Instead of performing an interpolation with Thiele’s formula, one can build a system of equations because the monomial degrees are known from the first interpolation.
5. Calculate the factors of numerator and denominator of $f(z_i; \{r_{j,2}\}; p_1)$ in z_i .
6. Compare the factors of $f(z_i; \{r_{j,1}\}; p_1)$ and $f(z_i; \{r_{j,2}\}; p_1)$ and accept only the coinciding factors. Bring these common factors to canonical form in order to reconstruct their monomial coefficients. The canonical polynomials are labeled as $q_{n/d}(z_i)$ for numerator (n) and denominator (d), respectively.
7. Repeat steps 1–6 over additional prime fields \mathbb{Z}_{p_k} and apply the Chinese Remainder Theorem [1] to combine the results until the rational reconstruction [17, 18] of the monomial coefficients of $q_{n/d}$ succeeds.

Note that there is a chance to randomly choose $\{r_{j,l}\}$ such that terms vanish. This probability is provided by the Zippel-Schwartz lemma [2, 51] and can be reduced by increasing the characteristic of the prime field.

While scanning for factors in z_i , we also determine its maximum degree. After all z_i are scanned, we perform an internal variable reordering with respect to the maximum degrees. The variable with maximum degree is set to one and only obtained via homogenization.

After a full scan, the obtained factors are used to cancel the corresponding terms of each black-box function. They are included again in the final result. Before returning the result to the user, the internal variable order is changed back to the user input.

To illustrate our algorithm, we perform the factor scan for the black-box function

$$f(z_1, z_2) = (z_1 - 1)(z_2 - 2)(z_1 z_2 - 1) = -2 + 2z_1 + z_2 + z_1 z_2 - 2z_1^2 z_2 - z_1 z_2^2 + z_1^2 z_2^2. \quad (30)$$

We start by scanning in z_1 , while setting the first prime field to \mathbb{Z}_{17} . The random number for z_2 is chosen to $z_2 = r_{2,1} = 5$. Interpolating the dependence of z_1 with distinct and randomized t and factorizing yields

$$f(z_1; \{5\}; 17) = 3 + 16z_1 + 15z_1^2 = 15(10 + z_1)(16 + z_1). \quad (31)$$

We now proceed by choosing $z_2 = r_{2,2} = 10$ which leads to

$$f(z_1; \{10\}; 17) = 8 + 14z_1 + 12z_1^2 = 12(5 + z_1)(16 + z_1). \quad (32)$$

By comparing Eq. (31) and Eq. (32), we identify the common factor $16 + z_1$ which is independent of the choice of z_2 . Therefore, we assume that this term is a correct factor in z_1 , which can be further processed by rational-reconstruction algorithms. Indeed, applying the same procedure over a different prime field and combining these results using the Chinese Remainder Theorem as well as rational reconstruction results in the factor

$$q_n(z_1) = z_1 - 1. \quad (33)$$

Analogously, we can find the factor $z_2 - 2$ in z_2 to arrive at the total factor

$$q_n(z_1, z_2) = (z_1 - 1)(z_2 - 2). \quad (34)$$

As mentioned before, this algorithm cannot find the mixed factor $z_1 z_2 - 1$ in Eq. (30). However, the remaining interpolation of $f(z_1, z_2)$ includes only this mixed term, when evaluating the black box as

$$\frac{f(y_1, y_2)}{q_n(y_1, y_2)} \quad (35)$$

at the values y_1 and y_2 .

In addition to the algorithm above, one could also apply a different approach by reconstructing rational functions in each variable univariately and only perform the factorization over \mathbb{Q} . Also this reconstruction has to be performed at least twice for random variable choices to identify the factors that are independent of the other variables. However, since

the interpolations are performed over fields with 63-bit characteristics, large coefficients can arise that require many prime fields to apply the rational reconstruction successfully, which leads to avoidable black-box probes. Furthermore, the factorization, which has to be computed after both reconstructions succeeded, takes additional computational effort as the numbers are not restricted to machine-size integers. Therefore, we do not provide an implementation of this alternative approach.

To quantify the improvements by this additional factor scan, we define the benchmark functions

$$f_1(z_1, \dots, z_4) = \frac{(1 + z_1 + z_2 + z_3 + z_4)^{20} - 1}{(1 + z_1 + z_2 + z_3 + z_4)^{20} - 1 + z_1^{20}}, \quad (36)$$

$$f_2(z_1, \dots, z_4) = (z_4 - 4)(z_4 - 3) \frac{(1 + z_1 + z_2 + z_3 + z_4)^{20} - 1}{(1 + z_1 + z_2 + z_3 + z_4)^{20} - 1 + z_1^{20}}, \quad (37)$$

$$f_3(z_1, \dots, z_{20}) = \frac{\sum_{i=1}^{20} z_i^{20}}{\sum_{i=1}^5 (z_1 z_2 + z_3 z_4 + z_5 z_6)^i}, \quad (38)$$

$$f_4(z_1, \dots, z_{20}) = \frac{\sum_{i=1}^{20} z_i^{20}}{\sum_{i=1}^5 (z_1 z_2 + z_3 z_4 + z_5 z_6)^i z_{20}^{35}}, \quad (39)$$

$$f_5(z_1, \dots, z_{20}) = \frac{\sum_{i=1}^{20} z_i^{20}}{\sum_{i=1}^5 (z_1 z_2 + z_3 z_4 + z_5 z_6)^i (z_2 z_{20}^{35} - 1)}. \quad (40)$$

The function f_1 is almost completely dense and no univariate factor is present. f_2 is the same as f_1 with two additional factors in z_4 . Factors of this type occur in many physical calculations with dimensional regularization. f_3 , f_4 , and f_5 are sparse functions with 20 variables. In f_4 the degree 35 term in z_{20} in the denominator factorizes and in f_5 a variable reordering ($z_1 \leftrightarrow z_{20}$) can simplify the interpolation. We compare the interpolation of the benchmark functions without and with the factor scan in Tab. 3. The scan for a sparse shift is enabled by default.

Table 3: Comparison of runtime and required probes for the benchmark functions defined by Eqs. (36)–(40) without and with the factor scan using a single thread of an AMD Ryzen 5 3400G processor.

Function	No factor scan		With factor scan		
	Probes	Runtime	Probes	Probes for scan	Runtime
f_1	42627	3.6 s	42959	332	3.8 s
f_2	50457	4.8 s	43050	432	3.8 s
f_3	1614	0.04 s	2504	890	0.25 s
f_4	38312	0.71 s	2539	925	0.25 s
f_5	38523	0.71 s	3930	923	0.35 s

For the dense function f_1 , our algorithm does not find any factor. Consequently, there is no improvement. However, the additional probes required to search for factors are negligible compared to the total number of probes. Additionally, the runtime is only affected mildly. When factors can be found, as is the case for f_2 , the interpolation can succeed faster and

with fewer probes, while only requiring about 100 additional probes for the scan compared to f_1 . The increased amount of probes during the scan can be explained by the higher degree in z_4 . For sparse functions as f_3 and f_4 , a possible improvement due to factorization depends strongly on the structure of the function. Since no factors in f_3 can be found by our algorithm, there is no improvement and the required probes increase by roughly 50% compared to the interpolation without the scan. However, when high-degree factors can be found as in f_4 , the number of black-box probes can decrease by an order of magnitude. Although no factors can be found by our algorithm for f_5 , reordering the variables leads to a significant improvement in the number of black-box probes by an order of magnitude. Since our shift-finder algorithm prefers variable shifts in the last variables, which are assumed to be of lower degree than the first variables, fewer additional monomials are generated by the shift, which further reduces the number of required black-box probes [19].

In our examples, the time for the black-box evaluations is negligible compared to the time for the interpolation. For physical applications, however, the evaluation time of the black box is usually dominant. Therefore, we consider a smaller number of probes as more important than the additional computational work required to search for factors. The impact of our factorization algorithm in a physical context is illustrated by an IBP reduction in Sect. 4.1.

3 Technical improvements and new features

In this section we describe technical improvements and new features implemented in **FireFly** 2.0. First, we changed the interface of **FireFly** according to the *curiously recurring template pattern* (CRTP), which allows us to compute probes with arrays of fixed size in order to reduce the overhead for some problems. The new interface is described in Sect. 3.1. In Sect. 3.2, we describe the new shunting-yard parser to read in expressions from files and strings in a way optimized for fast evaluations. Lastly, we implemented support for the parallelization with MPI. Our implementation and short instructions are described in Sect. 3.3.

3.1 New interface and overhead reduction

FireFly 2.0 introduces some changes in the black-box interface. The black-box functor has been extended to template arguments following the CRTP. This change allows for a static interface that can be used to evaluate the black box at sets of parameter points of different size. This new design can reduce a potential overhead of the black box by performing operations over vectors instead of scalars. To build the black box, one has to define a black-box functor, whose evaluation member function accepts a template argument, i.e.

```
class BlackBoxUser : public BlackBoxBase<BlackBoxUser> {
public:
    // Constructor
    BlackBoxUser(){};
    // Evaluation member function at the template argument FFIntTemp
```



```

template<typename FFIntTemp>
std::vector<FFIntTemp> operator()(const std::vector<FFIntTemp>& values) {
    // Define what the black box should do
    ...
}
// User-defined member functions
...
}

```

There are two changes in the interface shown in the above listing compared to the previous versions of **FireFly**. On the one hand, the definition of the functor is derived from its base class and takes itself as a template argument. This is the realization of the CRTP that allows for a static interface. On the other hand, the operator `()`, which is used to probe the black box at a given set of values is now templated. In practice, nothing changes for the user when these two modifications are implemented since the black box is only evaluated with objects that behave exactly like an **FFInt** as in previous versions of **FireFly**. These new objects are implemented in the form of the **FFIntVec** classes, which are fixed size arrays of **FFInt** objects. Currently, the sizes 2, 4, 8, 16, 32, 64, and 128 are implemented. Our implementation does not utilize the vectorization units of modern CPUs.

By default, the black box is only evaluated at a single parameter point and returns the evaluations of all functions in the black box at this parameter point. However, for some applications, like solving systems of equations, runtime improvements can be achieved when the system is solved once over an array of parameter points instead of several times over the individual parameter points. Therefore, the **Reconstructor** class can now be initialized with a maximum *bunch size* as

```

Reconstructor<BlackBoxUser> reconst(n_vars,
                                   n_threads,
                                   maximum_bunch_size,
                                   black_box);

```

When the third argument is greater than one, the **Reconstructor** class will call the evaluate function of the black-box functor with **FFIntVec** objects, i.e. collections of parameter points, up to the maximum bunch size. Each time a thread is assigned to compute a probe, it will gather probes to bunches and will then compute the probes over this bunch in a single evaluation. The size of the next bunch b is calculated by the formula

$$b = \min(2^{p_2}, b_{\max}) \quad \text{with} \quad p_2 = \max\left(0, \left\lfloor \log_2 \left(\frac{l_q}{t} \right) \right\rfloor\right) \quad (41)$$

for t threads, the maximum bunch size b_{\max} , and l_q probes in the queue. This design assures that a maximum amount of parallelism is favored over the reduction of potential overheads. Using large bunch sizes usually increases the required amount of memory significantly, e.g. each entry in a system of equation contains bunch size **FFInt** objects instead of one.

The parser described in Sect. 3.2 supports the evaluation in bunches and profits from it.

3.2 Parsing collections of rational functions

The first release version of **FireFly** expected the user to provide its implementation of the black box merely in native C++ code. To simplify the usage of **FireFly**'s interpolation algorithms, we implemented a parser, which reads in text files or strings of formulas and rewrites them into an optimized format for further evaluation. These formulas can then be used, for example, as black-box functions, or to create a new black box based on these expressions.

Only rational functions with rational numbers as coefficients are supported. They do not have to be provided in canonical form. The accepted notation of parsable functions is inspired by the syntax of **Mathematica** [52]. A file can contain several functions separated by semicolons `;`, which mark the end of each function. Variables can be composed of numbers and letters, i.e. elements of the set $\{a, b, \dots, z, A, B, \dots, Z, 0, 1, 2, \dots, 9\}$, with the restriction that a variable has to begin with a letter, e.g. `s12`. In addition, a variable is restricted to 16 characters. The supported binary arithmetic operators are

$$+, -, *, /, ^,$$

where `*` indicates a multiplication and `^` an exponentiation operation. Moreover, the unary operators `+` and `-` are supported and `+-` or `++` are interpreted as `-`. Implicit multiplication via the space character as well as any other combination of unary and binary operators are not supported. That is particularly important for negative exponents, e.g. x^{-10} , that are only interpreted correctly when set in parentheses.

To illustrate the usage of the parser, the file `fun` with the content

```
(-12 + 13/7*z1 + Z2^3 - (z1*Z2)^2)/z1;
```

shall be parsed. This can be done by including the header file `ShuntingYardParser.hpp` of the **FireFly** library and the following line of code:

```
firefly::ShuntingYardParser parser("fun", {"z1", "Z2"});
```

The `ShuntingYardParser` object is constructed with a path to a file that includes a collection of rational functions as the first argument and a vector of the occurring variables as strings as the second argument. During the construction of `parser`, the file `fun` is read in line by line and the string representing a function is joined up to the separator `;`. This string is then passed to an algorithm that rewrites the function to reverse Polish notation, also known as postfix notation, i.e. the function in the file `fun` becomes

$$-12 \ 13 \ 7 \ / \ z1 \ * \ + \ Z2 \ 3 \ ^ \ + \ z1 \ Z2 \ * \ 2 \ ^ \ - \ z1 \ /$$

and is stored inside `parser` as a vector of strings. The elements of this vector are referred to as tokens. A function in postfix notation can be evaluated in linear time with respect to the amount of tokens using a **stack**. The functions stored in `parser` can be evaluated at the values `y1` and `y2` by calling

```
auto res = parser.evaluate({y1, y2});
```

Note that `y1` and `y2` can be `FFInt` or `FFIntVec` objects. `res` is a vector of `FFInt` or `FFIntVec` objects depending on the choice of `y1` and `y2`, where each entry in `res` corresponds to the evaluation of a function found in the file `fun`. The position of the evaluated function corresponds to the position in the file. As `fun` only contains one function, `res` is a vector of size one in the example.

Instead of constructing the `ShuntingYardParser` object with a file, one can also pass a vector of functions in string format as the first argument. For example

```
firefly::ShuntingYardParser parser({{"z1+Z2"}, {"2*z1"}}, {"z1", "Z2"});
```

will construct `parser` with the two functions $z_1 + Z_2$ and $2z_1$.

When a collection of functions is expected to be evaluated several times over the same prime field, the `ShuntingYardParser` class provides the optimized version `evaluate_pre` of the member function `evaluate`, which makes use of pre-evaluated values for the tokens in the postfix notation. In practice, for the current prime field the images in \mathbb{Z}_p of the coefficients defined in \mathbb{Q} are calculated only once and stored for later use. Hence, rational numbers like $13/7$ are only evaluated once in \mathbb{Z}_p . Thus, it is advisable to store rational numbers as such, e.g. $13/7*x$ instead of $13*x/7$. Operators and variables are translated to an internal representation that is mapped to an integer value. These optimizations can lead to a significant performance improvement since the vector of strings is replaced by a vector of integers which is already evaluated except for the explicit values of the variables. To use this feature for the example above, one has to call:

```
parser.precompute_tokens();
auto res = parser.evaluate_pre({y1, y2});
```

The member function `precompute_tokens` has to be called only once any time the underlying prime field changes. Calling the constructor of a `ShuntingYardParser` object automatically executes `precompute_tokens`. `evaluate_pre` returns the same object as `evaluate`.

In order to demonstrate the performance of our `ShuntingYardParser`, we generate dense multivariate polynomials of three variables up to a degree bound D with monomial coefficients that are random rational numbers in the interval $[10^{98}, 10^{100}]$.¹ We also compare the performance without and with pre-evaluation. The results are shown in Tab. 4.

Our reference polynomial is of maximum degree $D = 92$ and, thus, consists of 138415 monomials. The file size of this polynomial is 30 MiB. The polynomials up to $D = 159$, $D = 200$, and $D = 252$ are chosen such that the number of monomials is roughly a factor 5, 10, and 20, respectively, larger than the reference polynomial to demonstrate the scaling of the parser. The file sizes scale correspondingly. As expected, the parsing and evaluation

¹We provide a `Mathematica` script that generates such polynomials in App. A

Table 4: Comparison of runtime and memory consumption of the `ShuntingYardParser` object without and with pre-evaluated tokens. These benchmarks were run on a single thread of an AMD Ryzen 5 3400G processor.

Function	No pre-evaluation		With pre-evaluation		Memory
	Parsing	Evaluation	Parsing	Evaluation	
$D = 92$	0.6 s	0.28 s	0.8 s	0.02 s	0.25 GiB
$D = 159$	2.5 s	1.4 s	3.9 s	0.11 s	1.2 GiB
$D = 200$	5.3 s	3.0 s	7.7 s	0.22 s	2.4 GiB
$D = 252$	11 s	6.0 s	16 s	0.45 s	4.8 GiB

times scale linearly in the number of monomials or, equivalently, in the number of operations. Although the parsing is up to 50 % slower when pre-evaluating values over a prime field, as one has to evaluate the function once, the evaluation time is an order of magnitude faster compared to no pre-evaluation.

Due to the linear scaling in the number of operations, polynomial representations that minimize this number, e.g. the Horner form, which is optimal in the number of operations, can lead to an additional improvement in the evaluation time. Therefore, the impact of the Horner representation of a polynomial is shown in Tab. 5 for the same set of functions in comparison.

Table 5: Comparison of runtime and memory consumption of the `ShuntingYardParser` object without and with pre-evaluated tokens pre-evaluated tokens using the Horner form. These benchmarks were run on a single thread of an AMD Ryzen 5 3400G processor.

Function	No pre-evaluation		With pre-evaluation		Memory
	Parsing	Evaluation	Parsing	Evaluation	
$D = 92$	0.4 s	0.21 s	0.6 s	3.7 ms	0.18 GiB
$D = 159$	2.2 s	1.1 s	3.0 s	17 ms	0.8 GiB
$D = 200$	4.5 s	2.1 s	5.9 s	32 ms	1.6 GiB
$D = 252$	9.0 s	4.2 s	12 s	62 ms	3.1 GiB

Utilizing this polynomial representation, the evaluation times can be decreased by an order of magnitude when using the pre-evaluation. Without pre-evaluation, there are still small performance gains. Hence, it is preferable to choose polynomial representations that minimize certain types of operations before evaluating them. Note that `FireFly` does not perform any optimization of polynomial representations except the pre-evaluation of tokens.

When parsing collections of functions, it is sometimes useful to omit redundant functions that might be evaluated several times. Especially during IBP reductions, many coefficients in the system of equations are the same and only have to be evaluated once. For such cases, the `ShuntingYardParser` provides the additional option which checks the parsed functions for duplicates. This check can be enabled when constructing a parser object with a third, but optional, argument:

```
firefly::ShuntingYardParser parser("fun", {"z1", "Z2"}, true);
```

By setting the third argument to `true`, the `ShuntingYardParser` evaluates the parsed functions for different randomized sets of variables and prime fields and searches for equal evaluations. Duplicate functions are removed and replaced by a key. Therefore, no function is evaluated more than once during a single evaluation step. The returned vector of the evaluation member functions remains unchanged. By default, no check for redundant functions is performed.

The parser also supports the evaluations in bunches described in Sect. 3.1. Tab. 6 shows the interpolation of the polynomial with $D = 200$ in Horner form using different bunch sizes. The CPU time for each probe drops by about 30 % and the total runtime still by 20 % when increasing the bunch size from 1 to 128. Since this is not a memory intensive problem after reading in the file, the memory consumption stays almost the same for all bunch sizes.

Table 6: Interpolation of the polynomial with $D = 200$ in Horner form using different maximal bunch sizes. These benchmarks were performed on cluster nodes equipped with two Intel Xeon Platinum 8160 processors with 24 cores each and hyperthreading disabled.

Bunch size	Runtime per probe	Total runtime
1	50 ms	6 h 30 min
2	47 ms	6 h 10 min
4	43 ms	5 h 50 min
8	40 ms	5 h 40 min
16	40 ms	5 h 40 min
32	37 ms	5 h 20 min
64	36 ms	5 h 20 min
128	34 ms	5 h 10 min

Lastly, `FireFly`'s parser is compatible with `Mathematica`'s `InputForm` stored to a file.

3.3 MPI

In addition to the parallelization with C++ threads, `FireFly` now supports parallelization with MPI, which enables the user to utilize multiple nodes on a computer cluster. We employ a master and worker pattern. Both master and worker processes use their own threadpools for the internal parallelization of each process and the communication between the individual processes is handled by the MPI protocol.

The master process runs the `Reconstructor` class as it would in the case without MPI, i.e. it handles the interpolation and reconstruction of the rational functions. If threads are idle, it also computes probes. The worker processes run the `MPIWorker` class which only computes probes. Each process has to provide a single thread to handle the communication with MPI. The master process distributes the required probes onto all processes available, including itself, and the worker processes send the results back when they are finished.

To use MPI an MPI library like `OpenMPI` [53,54] or `MPICH` [55] has to be installed. We advise the latter for optimal performance. Additionally, `FireFly` has to be compiled with the option `-DWITH_MPI=true`.

The basic code structure for the user is the following. First, the MPI environment has to be initialized. The thread support of MPI has to be set to `MPI_THREAD_SERIALIZED` or `MPI_THREAD_MULTIPLE`, because multiple threads can make MPI calls. However, we made sure that only one thread can make calls at a time. Thus, we advise to use the former for better performance. Then, the node has to find out whether it should run as the master or a worker process. In both cases, one has to define the black box. For the master process one creates the `Reconstructor` object and runs it exactly as in the case without MPI. For the worker processes, the user has to create `MPIWorker` objects. They are initialized similarly to the `Reconstructor` object and require the number of variables, the number of threads to use, the maximum bunch size, and the black box as arguments. The initialization of the `MPIWorker` objects already runs them. If the calculation of the `Reconstructor` on the master process finishes, they receive a signal to shut down. Finally, one has to finish the MPI environment.

```
// Initialization of MPI processes
int provided;
MPI_Init_thread(NULL, NULL, MPI_THREAD_SERIALIZED, &provided);

int process;
MPI_Comm_rank(MPI_COMM_WORLD, &process);

// Create the user-defined black box
BlackBoxUser black_box();

if (process == firefly::master) {
    firefly::Reconstructor<BlackBoxUser> reconst(n_vars,
                                                n_threads,
                                                maximum_bunch_size,
                                                black_box);

    reconst.reconstruct();
} else {
    firefly::MPIWorker<BlackBoxUser>(n_vars,
                                    n_threads,
                                    maximum_bunch_size,
                                    black_box);
}

// Finish MPI environment
MPI_Finalize();
```

We provide the file `example_mpi.cpp` as example. It is automatically compiled together with `FireFly` when setting `-DWITH_MPI=true`. After compiling, one can run the code with MPI by calling

```
mpiexec -n $NUMBER_OF_PROCESSES $EXECUTABLE
```

Since each MPI process has to allocate one thread for communication, one should only run one MPI process on each node and assign all threads of this node to it. For example, if two nodes with 48 threads each are available, running two MPI processes with 48 threads each yields a better performance than running four MPI processes with 24 threads each. For more command-line options of MPI, we refer to the respective manuals.

We again use the interpolation of the polynomial with $D = 200$ in Horner form from Sect. 3.2 to illustrate the performance gain by using several nodes of a computer cluster. Tab. 7 shows the scaling with 1, 2, and 4 nodes with 48 cores each. Using 2 nodes decreases the runtime by 26 % compared to a single node. However, doubling the nodes again only decreases it by 16 %. Thus, the scaling is not linear with the number of nodes. This is expected because the interpolation itself is performed on a single thread and is quite expensive for this dense high degree function. Only the computation of the probes utilizes all available threads. Moreover, they are very cheap individually for this synthetic benchmark as shown in Sect. 3.2. The performance gain is much higher for interpolation problems with a large number of expensive probes. One notable example are IBP reductions (s. Sect. 4) and we refer to the future publication [56] for the application of MPI in this context.

Table 7: Interpolation of the polynomial with $D = 200$ in Horner form from Sect. 3.2 using MPI on cluster nodes equipped with two Intel Xeon Platinum 8160 processors with 24 cores each and hyperthreading disabled.

Nodes	1	2	4
Total runtime	6 h 30 min	4 h 40 min	3 h 50 min

4 Applications in IBP reductions

During multi-loop calculations in high-energy physics one usually encounters large sums of L -loop scalar Feynman integrals

$$I(d, \{p_j\}, \{m_i\}, \{a_i\}) \equiv \int_{k_1, \dots, k_L} \frac{1}{P_1^{a_1} \dots P_N^{a_N}} \quad (42)$$

with

$$\int_k \equiv \int \frac{d^d k}{(2\pi)^d} \quad (43)$$

and the inverse propagators, i.e. $P_i = q_i^2 - m_i^2 + i\epsilon$ in Minkowski space. The q_i are linear combinations of the loop momenta k_l and the external momenta p_j . The integral $I(d, \{p_j\}, \{m_i\}, \{a_i\})$ depends on the space-time dimension d , the set of masses $\{m_i\}$, the set of external momenta $\{p_j\}$, and the propagator powers a_i which take integer values. The number of propagators N is restricted by L and the number of external momenta E to

$$N = EL + \frac{L(L+1)}{2}. \quad (44)$$

It is useful to define the sum of all positive powers of the propagators of an integral as

$$r \equiv \sum_{i=1}^N \theta \left(a_i - \frac{1}{2} \right) a_i \quad (45)$$

and the absolute value of the sum of all negative powers as

$$s \equiv \sum_{i=1}^N \theta \left(\frac{1}{2} - a_i \right) |a_i|, \quad (46)$$

where $\theta(x)$ is the Heaviside step function. Usually, an integral with higher r or higher s is regarded more difficult than an integral with lower r or s . Therefore, r and s can be used to sort the occurring integrals by difficulty.

Many of the scalar Feynman integrals given by Eq. (42) can be solved by the integration-by-parts (IBP) strategy of Chetyrkin and Tkachov [21, 22]. They observed that inserting the scalar product of a derivative with respect to a loop momentum with another momentum into Eq. (42) leads to a vanishing integral in dimensional regularization:

$$\int_{k_1, \dots, k_L} \frac{\partial}{\partial k_i^\mu} \left(\tilde{q}_j^\mu \frac{1}{P_1^{a_1} \dots P_N^{a_N}} \right) = 0, \quad (47)$$

where \tilde{q}_j^μ can either be another loop momentum or an external momentum. By explicitly evaluating the derivative one arrives at the linear relations

$$0 = \sum_n c_n I(d, \{p_j\}, \{m_i\}, \{a_i^{(n)}\}) \quad (48)$$

with modified $a_i^{(n)}$, where the values change by the addition or subtraction of small integers. The coefficients c_n are rational functions in d , $\{m_i\}$, and $\{p_j \cdot p_k\}$ with a small degree and also depend on the a_i in general. These relations are called IBP relations.

The most prominent strategy utilizing IBP relations is the Laporta algorithm [25]: By inserting integer values for the a_i in Eq. (48), one obtains a system of equations for the integrals. This system can be solved to express the integrals through a basis of master integrals, which have to be computed by other methods, see e.g. Ref. [57]. The Laporta algorithm has been implemented in the public codes **AIR** [26], **FIRE** [27–30], **Reduze** [31, 32], and **Kira** [33, 34]. However, the systems of equations for state-of-the-art calculations become huge and expensive to solve both in terms of memory and runtime, amongst other things due to large intermediate expressions.

Since the IBP relations are linear, the solution strategies only involve the addition, subtraction, multiplication, and division of the coefficients c_n . Therefore, the coefficients of the master integrals are also rational functions in d , $\{m_i\}$, and $\{p_j \cdot p_k\}$. Thus, the problems of the Laporta algorithm can be eased by finite-field techniques [35]. One can replace all occurring variables by elements of a finite field and solve the system of equations numerically, which is in general orders of magnitude faster than solving the system analytically. This can be used to identify and remove the linearly dependent equations before the actual analytic

reduction [36]. However, one can also replace the analytic solution by numerical solutions altogether by employing interpolation techniques [37]. The first public code implementing this strategy is FIRE6 [30], even though at least one private code exists for quite some time [58]. Based on these techniques, also new strategies for IBP reductions are pursued [39, 40]. For the related strategy based on generalized unitarity, the usage of more advanced techniques from computer science has been pioneered in Ref. [38].

We combined the integral-reduction program Kira [33, 34] with FireFly to utilize finite-field and interpolation techniques for IBP reductions and the improvements in FireFly 2.0. This new version of Kira will be made publicly available in a future publication, which will describe details of the implementation and provide sophisticated benchmarks [56]. Hence, we restrict ourselves to a short outlook in Sect. 4.1, which nonetheless makes the performance improvements in FireFly 2.0 evident.

The calculation of physical observables usually requires the insertion of the reduction table into a sum of Feynman integrals given by Eq. (42). For example, this has to be done to compute amplitudes as intermediate step of cross-section calculations. In these insertion steps, a problem similar to the reduction arises, because both the table and the expression can become huge for state-of-the-art problems. However, the final result expressed in terms of a sum of master integrals usually contains simpler coefficients compared to intermediate steps. Therefore, these problems can also be eased with the help of interpolation techniques over finite fields. The optimal strategy would be to interpolate the final coefficients without generating analytic reduction tables as intermediate step. Admittedly, these tables are still required for some strategies to compute the master integrals [57]. Thus, as a first step we developed a tool which can insert reduction tables into expressions using FireFly. It is described in Sect. 4.2.

4.1 IBP reduction with Kira and FireFly

To illustrate the performance improvements of FireFly 2.0 in physical problems, we perform the IBP reduction of the topology `topo5` shown in Fig. 1. We choose the physical

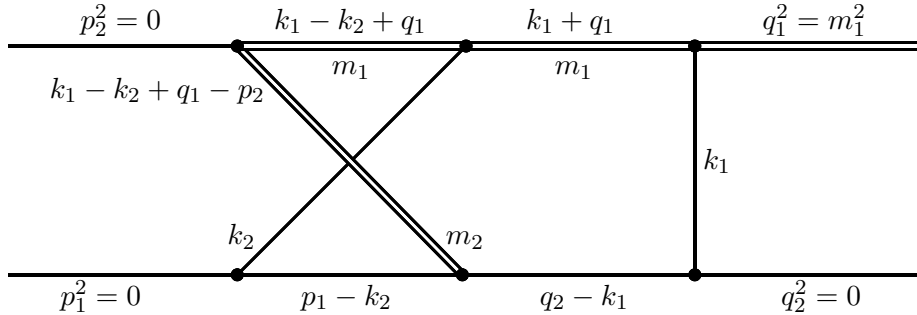


Figure 1: The non-planar double box `topo5` which occurs, e.g., in single top production. The diagram was produced with FeynGame [59].

value $r_{\max} = 7$, set the scale $m_1 = 1$, and use the option `select_mandatory_recursively` with the same range of r_{\max} and s_{\max} . We compare the latest development version of Kira both without and with FireFly 2.0. For the algebraic reduction, this version of Kira

contains only bug fixes and no performance improvements over Kira 1.2. In addition, we also include an older version of Kira which runs with FireFly 1.0. All of the versions are compiled with the g++ compiler in version 9.2.0 [60] and the standard memory allocator is replaced by linking jemalloc in version 5.2.1 [61] to the executables. Using an improved memory allocator leads to significant performance improvements for memory intensive computations like solving a system of equations on many threads. Both versions of FireFly use FLINT [49, 50] for the modular arithmetic. The algebraic version of Kira performs most of the computations with help of Fermat 6.31 [62]. These examples are solved on cluster nodes equipped with two Intel Xeon Platinum 8160 processors with 24 cores each and 192 GiB of RAM in total. Hyperthreading is disabled.

The results are shown in Tab. 8. Note that these numbers cannot directly be compared to

Table 8: Comparison of runtime and memory consumption for Kira, Kira with FireFly 1.0, and Kira with FireFly 2.0 in the reduction of `topo5` with the configuration described in the main text.

s_{\max}	Kira		Kira + FireFly 1.0		Kira + FireFly 2.0	
	Runtime	Memory	Runtime	Memory	Runtime	Memory
1	2 min 20 s	6.7 GiB	1 min 30 s	1.1 GiB	46 s	1.3 GiB
2	2 h	28 GiB	1 h	2.5 GiB	17 min	3.1 GiB
3	15 h 30 min	70 GiB	9 h	13 GiB	2 h 30 min	11 GiB
4	92 h	164 GiB	90 h	56 GiB	21 h	46 GiB

the numbers in Ref. [19], because different machines and memory allocators are used. The performance with FireFly 2.0 increases by a factor of 3–4 compared to FireFly 1.0. For higher s_{\max} , even the memory consumption decreases slightly due to the scan for univariate factors as discussed below.

Tab. 9 shows a detailed comparison of Kira with FireFly 1.0 and Kira with FireFly 2.0. The gain in runtime shown in Tab. 8 can be mainly attributed to the lower number of probes

Table 9: Detailed comparison of Kira with FireFly 1.0 and Kira with FireFly 2.0 in the reduction of `topo5` with the configuration described in the main text.

s_{\max}	Kira + FireFly 1.0			Kira + FireFly 2.0		
	Probes	CPU time per probe	CPU time for probes	Probes	CPU time per probe	CPU time for probes
1	15300	0.16 s	97 %	5200	0.14 s	95 %
2	330100	0.43 s	98 %	107500	0.40 s	97 %
3	1127000	0.95 s	92 %	433300	0.89 s	94 %
4	4304500	1.9 s	81 %	1624100	1.9 s	89 %

required, which decreased roughly by a factor of three throughout the whole range of s_{\max} . In addition, each probe became cheaper, because we replaced the parser of `pyRed` by our shunting-yard parser for which we use the pre-evaluation and removal of redundant functions as described in Sect. 3.2. With **FireFly 2.0**, the forward elimination dominates the whole process with about 94–95 %, whereas the initiation contributes with 3–4 % and the back substitution with only 1–2 %. Even though the new interpolation algorithms described in Sect. 2 are more complicated than the ones in **FireFly 1.0**, the CPU time for the probes still completely dominates the reduction.

The impact of the scan for univariate factors is shown in Tab. 10. It reduces the number

Table 10: Impact of the factor scan on the reduction of `topo5` with `Kira` in combination with **FireFly 2.0**. We use the configuration described in the main text.

s_{\max}	No factor scan			With factor scan		
	Runtime	Memory	Probes	Runtime	Memory	Probes
1	58 s	1.3 GiB	9000	46 s	1.3 GiB	5200
2	26 min	3.5 GiB	174000	17 min	3.1 GiB	107500
3	3 h 50 min	15 GiB	703700	2 h 30 min	11 GiB	433300
4	34 h	68 GiB	2745200	21 h	46 GiB	1624100

of probes and with it the runtime by roughly 40 %. In addition, it requires less memory especially for the more complicated reductions with higher s_{\max} . The main reason for these improvements is the fact that only one variable has to be shifted to generate constant terms for all coefficients. In contrast, three variables have to be shifted without the scan and, thus, more artificial terms have to be interpolated and stored. Without the scan, the memory consumption increases by more than 10 % compared to **FireFly 1.0**.

4.2 Insertion of IBP tables

The replacement tables obtained by IBP reductions are usually huge for state-of-the-art calculations. With increasing complexity of these results, utilizing algebraic tools like **Mathematica** [52] or **Fermat** [62] can become unfeasible in order to perform simplifications during the insertion into expressions like amplitudes. A more efficient approach can be to insert the IBP tables numerically and perform an interpolation of the final result over finite fields, e.g. with **FireFly**. In order to automatize the insertion step and to illustrate applications of interpolation algorithms, **FireFly 2.0** provides the additional executable `ff_insert`, which is compiled by default. `ff_insert` can be found in the `build` directory or, when installed, in the `bin` directory inside the installation directory. Its usage requires the user to provide the directories `config` and `replacements` containing all required files. The directory structure is as follows:

```
config/
|__ vars
```

```

|__ functions
|__ skip_functions
replacements/
|__ $REPLACEMENT_LIST_1
|__ ...

```

The **vars** and **functions** files contain the list of occurring variables and the list of integral families, respectively, separated by new lines. In the **skip_functions** file one can specify master integrals that should be omitted during the insertion. In the **replacements** directory, several files with arbitrary file name can be stored containing replacement rules for integrals in **Mathematica** syntax. **FireFly** expects the formulas in a format, where the integral appears first followed by a potential coefficient, e.g.,

$$F1[1,0,1,-2]*(s+t+d)/42 + F1[1,1,1,1]*(d-3),$$

where **F1** is an integral family defined in **functions** and **s**, **t**, and **d** are variables defined in **vars**.

The insertion of the replacement rules into the expression **\$INPUT_FILE** can be started by calling

```
ff_insert $INPUT
```

in the directory, where the **config** and **replacements** directories can be found. The last command line option (**\$INPUT**) should be an input file, e.g. an amplitude, or a directory containing several expressions for which replacements should be performed. The expressions have to be provided in the same syntax as the replacement tables. The following options can be specified:

```

-p $NUMBER_OF_THREADS // Defines the number of threads
-bs $MAXIMUM_BUNCH_SIZE // Defines the maximum bunch size
-m // Merge expressions in the given directory
-nfs // Disables the factor scan
-ni // Stores the unsimplified coefficients
-s // Enables saving of states
-h // Show options

```

FireFly will read in the expression in **\$INPUT** first. Afterwards, the replacement tables are parsed and only the required replacement rules are selected. **FireFly** does not support repeated replacements. They can be obtained by either preparing the tables or by reusing the **ff_insert** executable multiple times. Afterwards, the replacements are performed and a list of remaining integrals is created, which are assumed to be master integrals. Each coefficient of a master integral is passed to **FireFly**'s parser and then used for the interpolation. The functional interpolations are performed sequentially, i.e. each coefficient of a master integral is interpolated independently of the others. When all interpolations succeeded, the file **out.\$INPUT** is written to the current directory. It contains a sum of master integrals and their respective coefficients. When only performing simplifications of master-integral coefficients, the **replacements** directory is not mandatory.

Since the interpolation of each master integral coefficient can be performed irrespective of the others, one can in principle use separate machines for each function. Therefore, the option `-ni` indicates that no interpolation has to be performed. Instead, all unsimplified coefficients are written to files in the newly created `coefficients` directory. They can be used as input for the `ff_insert` executable or other processing.

We provide example data for an insertion job to be run with the `ff_insert` executable in the `examples` directory of the `FireFly` files.

5 Conclusions

We described the main improvements implemented into `FireFly` since the release last year. On the algorithmic side, we implemented the racing algorithm of Refs. [6, 7] to improve the univariate interpolation of sparse polynomials as sub-algorithm for the interpolation of rational functions. Moreover, we described the hybrid racer algorithm for rational functions in Sect. 2.2, which removes the oversampling for dense functions compared to the algorithm of Ref. [15]. Since the results of physical problems often contain simple factors as shown in Refs. [23, 24], we implemented an additional scan to search for univariate factors to remove them in the actual interpolation.

On the technical side, we changed the interface of `FireFly` to allow for a reduction of overhead by computing several probes in a single evaluation. We also implemented a shunting-yard parser for expressions stored in files and strings, which can be used to construct black boxes. Moreover, `FireFly` now supports MPI in order to utilize computer clusters.

The impact of the algorithmic improvements on a physical IBP reduction has been studied in Sect. 4.1 by combining the program `Kira` with `FireFly`. In this context, we also provide a tool based on `FireFly`, which allows to insert replacement tables into expressions. The new version of `Kira` supporting reductions with `FireFly` will become publicly available accompanied by a separate publication [56].

`FireFly` 2.0 is publicly available at

<https://gitlab.com/firefly-library/firefly> .

Acknowledgments

We thank Philipp Maierhöfer for the idea of overhead reductions with bunches and discussions about this topic. We thank Johann Usovitsch for inspiring the factor scan and discussions about this topic. Furthermore, we would like to thank Long Chen, Robert Harlander, Philipp Maierhöfer, and Johann Usovitsch for comments on the manuscript. We thank Herschel Chawdhry, Long Chen, Joshua Davies, Jerry Dormans, Philipp Maierhöfer, Mario Prausa, Robert Schabinger, and Johann Usovitsch for bug reports and feature suggestions.

This research was supported by the *Deutsche Forschungsgemeinschaft* (DFG, German Research Foundation) under grant 396021762 – TRR 257. F.L. acknowledges financial support

by DFG through project [386986591](#). Some of the calculations were performed with computing resources granted by RWTH Aachen University under project `rwth0541`.

A Mathematica code to generate dense polynomials

In this appendix we provide Mathematica functions that generate random polynomials in either Horner or expanded form. They read

```
polygenHorner[{x_,y_,z_}, degree_]:= HornerForm[Sum[
  RandomInteger[{1,100}]/RandomInteger[{1,100}]*x^i*y^j*z^k
  , {i, 0, degree}, {j, 0, degree - i}, {k, 0, degree - i - j}
]];
polygen[{x_,y_,z_}, degree_]:= Sum[
  RandomInteger[{1,100}]/RandomInteger[{1,100}]*x^i*y^j*z^k
  , {i, 0, degree}, {j, 0, degree - i}, {k, 0, degree - i - j}
];
```

and generate polynomials of three variables, x, y, and z, up to a given degree. When performing the replacements

```
rep1 = {Rational[a_,b_] -> Rational[RandomInteger[{a*10^98,a*10^99}],
  RandomInteger[{b*10^98,b*10^99}]]};
rep2 = {Plus[a_Integer,b_] -> Plus[
  Rational[RandomInteger[{a*10^98,a*10^99}],
  RandomInteger[{3*a*10^98, 3*a*10^99}]],b]};
```

the coefficients of each monomial are randomized rational numbers in the range of $[10^{98}, 10^{100}]$. The obtained polynomials which are not in Horner form are then rewritten to a format, where the rational number is in front of the monomial.

References

- [1] J. von zur Gathen, J. Gerhard, *Modern Computer Algebra*, third ed., Cambridge University Press, 2013.
- [2] R. Zippel, *Probabilistic algorithms for sparse polynomials*, *Symbolic Algebraic Comp. EUROSAM* **1979** (1979) 216–226.
- [3] R. Zippel, *Interpolating Polynomials from their Values*, *J. Symb. Comp.* **9** (1990) 375–403.
- [4] M. Ben-Or, P. Tiwari, *A Deterministic Algorithm for Sparse Multivariate Polynomial Interpolation*, *Proc. ACM Symp. Theory Comp.* **20** (1988) 301–309.
- [5] E. Kaltofen, Lakshman Y., *Improved Sparse Multivariate Polynomial Interpolation Algorithms*, *Symbolic Algebraic Comp. ISSAC* **1988** (1989) 467–474.
- [6] E. Kaltofen, W.-s. Lee, A.A. Lobo, *Early Termination in Ben-Or/Tiwari Sparse Interpolation and a Hybrid of Zippel’s Algorithm*, *Proc. Int. Symp. Symbolic Algebraic Comp.* **2000** (2000) 192–201.
- [7] E. Kaltofen, W.-s. Lee, *Early termination in sparse interpolation algorithms*, *J. Symb. Comp.* **36** (2003) 365–400.
- [8] S.M.M. Javadi, M. Monagan, *Parallel Sparse Polynomial Interpolation over Finite Fields*, *Proc. Int. Workshop Parallel Symbolic Comp.* **4** (2010) 160–168.
- [9] E. Kaltofen, B.M. Trager, *Computing with Polynomials Given by Black Boxes for Their Evaluations: Greatest Common Divisors, Factorization, Separation of Numerators and Denominators*, *J. Symb. Comp.* **9** (1990) 301–320.
- [10] D.Y. Grigoriev, M. Karpinski, M.F. Singer, *Interpolation of Sparse Rational Functions Without Knowing Bounds on Exponents*, *Proc. Symp. Foundations Comp. Sci.* **31** (1990) 840–846.
- [11] D.Y. Grigoriev, M. Karpinski, *Algorithms for Sparse Rational Interpolation*, *Proc. Int. Symp. Symbolic Algebraic Comp.* **1991** (1991) 7–13.
- [12] D. Grigoriev, M. Karpinski, M.F. Singer, *Computational Complexity of Sparse Rational Interpolation*, *SIAM J. Comp.* **23** (1994) 1–11.
- [13] J. de Kleine, M. Monagan, A. Wittkopf, *Algorithms for the Non-monic Case of the Sparse Modular GCD Algorithm*, *Proc. Int. Symp. Symbolic Algebraic Comp.* **2005** (2005) 124–131.
- [14] E. Kaltofen, Z. Yang, *On Exact and Approximate Interpolation of Sparse Rational Functions*, *Proc. Int. Symp. Symbolic Algebraic Comp.* **2007** (2007) 203–210.
- [15] A. Cuyt, W.-s. Lee, *Sparse interpolation of multivariate rational functions*, *Theor. Comp. Sci.* **412** (2011) 1445–1456.
- [16] Q.-L. Huang, X.-S. Gao, *Sparse Rational Function Interpolation with Finitely Many*

- Values for the Coefficients, *Math. Aspects Comp. Information Sci.* **2017** (2017) 227–242, [arXiv:1706.00914 \[cs.SC\]](#).
- [17] P.S. Wang, *A p-adic Algorithm for Univariate Partial Fractions*, *Proc. ACM Symp. Symbolic Algebraic Comp.* **1981** (1981) 212–217.
 - [18] M. Monagan, *Maximal Quotient Rational Reconstruction: An Almost Optimal Algorithm for Rational Reconstruction*, *Proc. Int. Symp. Symbolic Algebraic Comp.* **2004** (2004) 243–249.
 - [19] J. Klappert, F. Lange, *Reconstructing Rational Functions with FireFly*, *Comp. Phys. Commun.* **247** (2020) 106951, [arXiv:1904.00009 \[hep-ph\]](#).
 - [20] T. Peraro, *FiniteFlow: multivariate functional reconstruction using finite fields and dataflow graphs*, *JHEP* **1907** (2019) 031, [arXiv:1905.08019 \[hep-ph\]](#).
 - [21] F.V. Tkachov, *A theorem on analytical calculability of 4-loop renormalization group functions*, *Phys. Lett. B* **100** (1981) 65–68.
 - [22] K.G. Chetyrkin, F.V. Tkachov, *Integration By Parts: The algorithm to calculate beta functions in four loops*, *Nucl. Phys. B* **192** (1981) 159–204.
 - [23] A.V. Smirnov, V.A. Smirnov, *How to choose master integrals*, [arXiv:2002.08042 \[hep-ph\]](#).
 - [24] J. Usovitsch, *Factorization of denominators in integration-by-parts reductions*, [arXiv:2002.08173 \[hep-ph\]](#).
 - [25] S. Laporta, *High-precision calculation of multi-loop Feynman integrals by difference equations*, *Int. J. Mod. Phys. A* **15** (2000) 5087–5159, [hep-ph/0102033](#).
 - [26] C. Anastasiou, A. Lazopoulos, *Automatic Integral Reduction for Higher Order Perturbative Calculations*, *JHEP* **0407** (2004) 046, [hep-ph/0404258](#).
 - [27] A.V. Smirnov, *Algorithm FIRE – Feynman Integral REduction*, *JHEP* **0810** (2008) 107, [arXiv:0807.3243 \[hep-ph\]](#).
 - [28] A.V. Smirnov, V.A. Smirnov, *FIRE4, LiteRed and accompanying tools to solve integration by parts relations*, *Comp. Phys. Commun.* **184** (2013) 2820–2827, [arXiv:1302.5885 \[hep-ph\]](#).
 - [29] A.V. Smirnov, *FIRE5: a C++ implementation of Feynman Integral REduction*, *Comp. Phys. Commun.* **189** (2015) 182–191, [arXiv:1408.2372 \[hep-ph\]](#).
 - [30] A.V. Smirnov, F.S. Chukharev, *FIRE6: Feynman Integral REduction with Modular Arithmetic*, *Comp. Phys. Commun.* **247** (2020) 106877, [arXiv:1901.07808 \[hep-ph\]](#).
 - [31] C. Studerus, *Reduze - Feynman Integral Reduction in C++*, *Comp. Phys. Commun.* **181** (2010) 1293–1300, [arXiv:0912.2546 \[physics.comp-ph\]](#).
 - [32] A. von Manteuffel, C. Studerus, *Reduze 2 - Distributed Feynman Integral Reduction*, [arXiv:1201.4330 \[hep-ph\]](#).

- [33] P. Maierhöfer, J. Usovitsch, P. Uwer, *Kira - A Feynman Integral Reduction Program*, *Comp. Phys. Commun.* **230** (2018) 99–112, [arXiv:1705.05610 \[hep-ph\]](#).
- [34] P. Maierhöfer, J. Usovitsch, *Kira 1.2 Release Notes*, [arXiv:1812.01491 \[hep-ph\]](#).
- [35] M. Kauers, *Fast solvers for dense linear systems*, *Nucl. Phys. Proc. Suppl.* **183** (2008) 245–250.
- [36] P. Kant, *Finding Linear Dependencies in Integration-By-Parts Equations: A Monte Carlo Approach*, *Comp. Phys. Commun.* **185** (2014) 1473–1476, [arXiv:1309.7287 \[hep-ph\]](#).
- [37] A. von Manteuffel, R.M. Schabinger, *A novel approach to integration by parts reduction*, *Phys. Lett. B* **744** (2015) 101–104, [arXiv:1406.4513 \[hep-ph\]](#).
- [38] T. Peraro, *Scattering amplitudes over finite fields and multivariate functional reconstruction*, *JHEP* **1612** (2016) 030, [arXiv:1608.01902 \[hep-ph\]](#).
- [39] D. Bendle, J. Böhm, W. Decker, A. Georgoudis, F.-J. Pfreundt, M. Rahn, P. Wasser, Y. Zhang, *Integration-by-parts reductions of Feynman integrals using Singular and GPI-Space*, *JHEP* **02** (2020) 079, [arXiv:1908.04301 \[hep-th\]](#).
- [40] X. Guan, X. Liu, Y.-Q. Ma, *Complete reduction of two-loop five-light-parton scattering amplitudes*, [arXiv:1912.09294 \[hep-ph\]](#).
- [41] G. De Laurentis, D. Maître, *Extracting analytical one-loop amplitudes from numerical evaluations*, *JHEP* **07** (2019) 123, [arXiv:1904.04067 \[hep-ph\]](#).
- [42] G. De Laurentis, *Analytical amplitudes from numerical solutions of the scattering equations*, *JHEP* **02** (2020) 194, [arXiv:1910.11355 \[hep-ph\]](#).
- [43] J. Hu, M. Monagan, *A Fast Parallel Sparse Polynomial GCD Algorithm*, *Proc. Int. Symp. Symbolic Algebraic Comp.* **2016** (2016) 271–278.
- [44] E.R. Berlekamp, *Algebraic Coding Theory*, second ed., Aegean Park Press, 1984.
- [45] J. Massey, *Shift-Register Synthesis and BCH Decoding*, *IEEE Trans. Inf. Theorie* **15** (1969) 122–127.
- [46] K. Imamura, W. Yoshida, *A Simple Derivation of the Berlekamp-Massey Algorithm and Some Applications*, *IEEE Trans. Inf. Theorie* **33** (1987) 146–150.
- [47] E. Jonckheere, C. Ma, *A Simple Hankel Interpretation of the Berlekamp-Massey Algorithm*, *Linear Algebra Its Appl.* **125** (1989) 65–76.
- [48] A. Díaz, E. Kaltofen, *FOXBOX: A System for Manipulating Symbolic Objects in Black Box Representation*, *Proc. Int. Symp. Symbolic Algebraic Comp.* **1998** (1998) 30–37.
- [49] W. Hart et al., *FLINT: Fast Library for Number Theory*, <http://www.flintlib.org/>.
- [50] W.B. Hart, *Fast Library for Number Theory: An Introduction*, *Proc. Int. Congress Mathe. Software* **2010** (2010) 88–91.
- [51] J.T. Schwartz, *Fast Probabilistic Algorithms for Verification of Polynomial Identities*, *J. ACM* **27** (1980) 701–717.

- [52] Wolfram Research, Inc., *Mathematica Version 11.3*.
- [53] E. Gabriel et al., *Open Source High Performance Computing*, <https://www.open-mpi.org/>.
- [54] E. Gabriel et al., *Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation*, *Adv. Parallel Virtual Machine Message Passing Interface. EuroPVM/MPI 2004* (2004) 97–104.
- [55] W. Gropp et al., *MPICH*, <https://www.mpich.org/>.
- [56] J. Klappert, F. Lange, P. Maierhöfer, J. Usovitsch, in preparation.
- [57] V.A. Smirnov, *Analytic Tools for Feynman Integrals*, first ed., Springer-Verlag Berlin Heidelberg, 2012.
- [58] A. von Manteuffel, R.M. Schabinger, *Quark and gluon form factors to four-loop order in QCD: the N_f^3 contributions*, *Phys. Rev. D* **95** (2017) 034030, [arXiv:1611.00795 \[hep-ph\]](https://arxiv.org/abs/1611.00795).
- [59] R.V. Harlander, S.Y. Klein, M. Lipp, *FeynGame*, [arXiv:2003.00896 \[physics.ed-ph\]](https://arxiv.org/abs/2003.00896).
- [60] Analog Devices et al., *GCC, the GNU Compiler Collection*, <https://gcc.gnu.org/>.
- [61] J. Evans et al., *jemalloc – memory allocator*, <http://jemalloc.net/>.
- [62] R.H. Lewis, *Fermat: A Computer Algebra System for Polynomial and Matrix Computation*, <http://home.bway.net/lewis/>.