TTK-19-11
P3H-19-007

Reconstructing Rational Functions with `FireFly`

Jonas Klappert and Fabian Lange

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

We present the open-source `C++` library `FireFly` for the reconstruction of multivariate rational functions over finite fields. We discuss the involved algorithms and their implementation. As an application, we use `FireFly` in the context of integration-by-parts reductions and compare runtime and memory consumption to a fully algebraic approach with the program `Kira`.

Contents

1 Introduction

The interpolation of polynomials has been researched since the 18th century and some of the algorithms, e.g. Newton's interpolation algorithm [1], are still used today. They use black-box probes, i.e. evaluations at numerical values, of the to be interpolated function, the black box, for the interpolation. However, these original algorithms were designed for univariate polynomials and are costly to generalize to the multivariate case. Interpolation algorithms specific for multivariate polynomials have been studied intensively for several decades, e.g. in Refs. [2–9].

The interpolation of rational functions is a younger field of research, even though Thiele's univariate interpolation formula [1] has been known for more than one hundred years. To our knowledge, the first multivariate algorithm came up in the early 1990s. Grigoriev, Karpinski, and Singer presented an algorithm to obtain a bound on the degrees of numerator and denominator, then build a system of equations, and solve it for all possible coefficients [10–12]. This is a dense algorithm, i.e. it assumes that all coefficients of the rational function up to the bounds are non-zero. The next algorithm was presented by Khodadad and Monagan in Ref. [13] in 2006 and is based on the Extended Euclidean Algorithm [14]. In the following year, Kaltofen and Zhang published an algorithm based on the separate evaluation of numerator and denominator [15]. It also exploits the sparsity of the rational function if only a small subset of possible coefficients is non-zero. The algorithm by Cuyt and Lee from 2010 first performs a univariate interpolation using Thiele's formula and then uses the coefficients as input for a multivariate polynomial interpolation [16]. In 2017, Huang and Gao [17] presented an algorithm which uses Kronecker substitution [14].

Most of the algorithms mentioned above rely on finite field arithmetic. Finite fields, e.g. integers modulo a prime number, are used for several hundred years. These calculations are done in an exact arithmetic environment which, however, avoids a number swell. This means that the coefficients of the polynomials and rational functions obtained by the interpolation algorithms are numbers in the finite field. In 1981, Wang presented an algorithm based on the Extended Euclidean Algorithm [14] which allows to reconstruct rational numbers from their image in a finite field [18].

We review finite fields, interpolation algorithms for both polynomials and rational functions, and the reconstruction algorithms for rational numbers in Sect. 2. The `C++` library `FireFly` uses some of these algorithms. For the rational reconstruction we employ the algorithm by Cuyt and Lee [16] combined with Zippel's algorithm for the polynomial interpolation [2, 3]. We also present some modifications to reduce the number of black-box probes. In Sect. 3 we present the implementation of `FireFly` and show how it can be used. We also present some benchmarks for the interpolation of multivariate rational functions.

As an application we apply `FireFly` to the reduction of Feynman integrals with integration-by-parts (IBP) relations [19, 20], which are a standard calculation technique in multi-loop calculations of theoretical particle physics. They relate different integrals with each other through linear relations. The first solution strategy is to rearrange the IBP relations to recursion relations, which always express an integral through easier integrals. Applying the recursion relations repeatedly allows to express all integrals through master integrals. At

three-loop level, this approach has been successfully applied for massless propagator-type diagrams [20–22], massive tadpoles [23, 24], and on-shell propagators [25–27]. However, at higher orders in perturbation theory it becomes unpractical to obtain such recursion relations. Nonetheless, some progress in deriving such relations automatically has been made in the last decade [28–31].

The second strategy was presented by Laporta in 2001 [32]. He suggested to build a system of equations out of the IBP relations by inserting values for the propagator powers and then solve this system. The Laporta algorithm and its modifications have been implemented in several public codes, `AIR` [33], `FIRE` [29, 34–36], `Reduze` [37, 38], and `Kira` [39, 40], as well as in numerous private codes. However, the systems of equations can become gigantic and, thus, expensive to solve both in terms of memory and runtime, which is partly related to large intermediate expressions.

To circumvent these problems, the use of finite-field techniques has been proposed by Kauers in 2008 together with a `Mathematica` package as an example [41]. In 2013, Kant proposed to solve the system of equations over a finite field before running the Laporta algorithm in order to reduce the size of the system by identifying and removing the linearly dependent equations [42]. This auxiliary use of finite fields has been implemented in `Kira` and led to significantly improved runtimes [39]. The use of the interpolation techniques over finite fields in the context of the Laporta algorithm has been advocated in Ref. [43]. In 2016, Peraro summarized some of the interpolation techniques from computer science in the context of generalized unitarity [44]. The first calculation using finite-field interpolation techniques for IBP reductions was accomplished by von Manteuffel and Schabinger in 2016 [45]. This was a one scale problem and, thus, also a one variable problem. Recently, three more one-scale calculations have been finished [46–48]. The recently published version 1.2 of `Kira` uses a multivariate Newton interpolation as supportive technique [40], albeit over $\mathbb{Z}$ instead of a finite field. Shortly after, `FIRE6` was published as first public implementation of a Laporta algorithm with a multivariate interpolation of rational functions over a finite field [36]. It supports the stable interpolation with two variables, i.e. two scales when setting one of them to one.

After a concise review of IBP reductions, we briefly describe our implementation with `FireFly` and compare it with the traditional reduction with `Kira` for some examples in Sect. 4. We also point out the advantages and disadvantages of the finite-fields-interpolation approach.

Another important method for multi-loop calculations is based on generalized unitarity. As mentioned above, Peraro pioneered the application of finite-field interpolation techniques for this method in 2016 [44]. This lead to several successful calculations in the last two years [49–56], where functions with up to four variables have been interpolated. However, we do not delve into this field in our paper.

2 Functional interpolation over finite fields

The interpolation of a function f of n variables $\vec{z} = (z_1, \dots z_n)$ is based on the idea to require no knowledge about the function itself and only use its evaluations at different values of $\vec{z}$ to construct an analytic form of the probed function or at least find an approximation formula. These kinds of interpolation problems are also called black-box interpolation problems, where the function to be interpolated serves as the black box. These interpolations are performed over a field. A set of particularly suited fields, due to their specific properties, are finite fields $\mathbb{Z}_p$ with characteristic p, where p is the defining prime, on which we will focus in this paper. All calculations are thus carried out modulo p avoiding number swell and thus saving memory and runtime. The multiplicative inverse is unique and can be determined using the Extended Euclidean Algorithm [14].

In the following sections, we describe how polynomials and rational functions of in principle arbitrarily many variables can be efficiently interpolated and how one can promote elements of $\mathbb{Z}_p$ to the field of rational numbers $\mathbb{Q}$.

2.1 Interpolation of polynomials

To fix the notation, we start by defining multivariate polynomials as follows. Given a set of n variables $\vec{z} = (z_1, \dots z_n)$ and an n-dimensional multi-index $\alpha = (\alpha[1], \dots, \alpha[n])$ containing integers $\alpha[i] \geq 0$, we define a monomial $\vec{z}^{\,\alpha}$ as

$$\vec{z}^{\,\alpha} \equiv \prod_{i=1}^{n} z_i^{\alpha[i]} \tag{1}$$

with a degree r of

$$r = \sum_{i=1}^{n} \alpha[i]. \tag{2}$$

A polynomial f, which is a member of the polynomial ring $\mathbb{Z}_p[\vec{z}]$ in the variables $\vec{z}$, is defined as

$$f(\vec{z}) = \sum_{j} c_{\alpha_j} \vec{z}^{\,\alpha_j}. \tag{3}$$

The coefficients c_{α_j} are elements of $\mathbb{Z}_p$ corresponding to different multi-indices α_j.

Solving the black-box interpolation problem of a multivariate polynomial can be done by recursive interpolations of univariate polynomials. Thus, we briefly mention how to interpolate univariate polynomials before turning to the multivariate case. A well-known interpolation algorithm for univariate polynomial functions $f = f(z_1) \in \mathbb{Z}_p[z_1]$ is given by the Newton interpolation which relies on Newton polynomials [1]. Given a sequence of distinct interpolation points $y_{1,1}, \dots, y_{1,R+1} \in \mathbb{Z}_p$, the Newton polynomial for f of degree R can be written as

$$f(z_1) = a_0 + \sum_{i=1}^{R} a_i \prod_{j=1}^{i} (z_1 - y_{1,j+1}), \tag{4}$$

where the coefficients a_i can be recursively defined as

$$a_i \equiv a_{i,i}, \tag{5}$$

$$a_{i,j} = \frac{a_{i,j-1} - a_{j-1}}{y_{1,i+1} - y_{1,j}}, \tag{6}$$

$$a_{i,0} = f(y_{1,i+1}). \tag{7}$$

A benefit of this algorithm is that the calculation of a new term does not alter the previously computed terms. This feature makes the Newton interpolation particularly suited for interpolations with an unknown degree R. The interpolation terminates if an a_i is equal to zero. As a check, one could evaluate further a_i with additional $y_{1,i}$ which should also be zero.

There are two caveats which have to be mentioned. First, one has to be careful of spurious poles which can occur in the recursion formula. In this case one has to choose another value of $y_{1,i}$ and proceed with the algorithm. To minimize the chance of spurious poles, we use random numbers for $y_{1,i}$ and fields with a large characteristic. Secondly, the termination criterion of finding $a_i = 0$ is just a probabilistic one when the interpolation is performed in $\mathbb{Z}_p$. Since there is only a finite number of elements, one could accidentally find a zero and the interpolation of f terminates wrongly. The probability can be reduced by calculating a_i multiple times for different random choices of $y_{1,i}$ and by choosing a larger a characteristic for $\mathbb{Z}_p$. In our studies, no wrongly terminated interpolation was found when defining $\mathbb{Z}_p$ with the largest 63-bit primes. Hence, for practical usage, it might be sufficient to omit additional checks for a_i after one encounters $a_i = 0$.

The algorithm can be easily generalized to the case of multivariate polynomials by a recursive application of the Newton interpolation for univariate functions, e.g. as proposed in Ref. [44]. Consider a generic multivariate polynomial $f \in \mathbb{Z}_p[z_1, \ldots, z_n]$. f can be reinterpreted as a univariate polynomial in z_1 of degree R, with its coefficients being multivariate polynomials of $n-1$ variables $(z_2, \ldots, z_n)$. This promotes the coefficients a_i to be elements of the polynomial ring $\mathbb{Z}_p[z_2, \ldots z_n]$ such that

$$f(z_1, \ldots, z_n) = \sum_{i=0}^{R} a_i(z_2, \ldots, z_n) \prod_{j=1}^{i} (z_1 - y_{1,j}), \tag{8}$$

where $y_{1,i} \in \mathbb{Z}_p$ are randomly chosen interpolation points for z_1 as in the univariate case. Obviously, one can again use the same partition for $a_i(z_2, \ldots, z_n)$ leading to new coefficients which are functions of $n-2$ arguments. After applying this procedure $n-1$ times, we are left with the univariate interpolation problem, which can be solved by the Newton interpolation. For later usage, we denote the numerical choices $y_{i,j}$ for z_i as the corresponding value to z_i at order j. The orders of each variable form a tuple which we call z_i order, i.e. the numerical choices $y_{2,2}$, $y_{3,3}$ form the z_i order $(2, 3)$.

For practicality, it is useful to transform the interpolated Newton polynomial to its canonical form defined in Eq. (3). This can be done by using additions of multivariate polynomials and multiplications of a multivariate polynomial by a linear univariate polynomial.

However, the Newton interpolation algorithm is a *dense* algorithm, i.e. it interpolates all possible terms of a polynomial even if it is *sparse* and only a small subset of coefficients

is non-zero. It requires $\prod_i (R_i + 2)$ black-box probes of the polynomial $f(z_1, \ldots, z_n)$ in general, where R_i is the maximal degree of z_i. Overall, the complexity of this algorithm is exponential in n and grows quickly as the number of variables and the individual maximal degrees increase, which is inefficient for sparse polynomials. Note that a degree bound on each variable can reduce the complexity to be $\prod_i (R_i + 1)$. This is still not preferable for sparse polynomials which are usually encountered in physical calculations.

A more efficient multivariate algorithm in terms of black-box probes is the Zippel algorithm [2,3], which in addition takes advantage of the sparsity of the polynomial. As in the previous algorithm, Zippel's algorithm interpolates one variable at a time. The interpolation of a variable is called a *stage*, i.e. stage one is the interpolation $f(z_1, y_{2,1}, y_{3,1}, \ldots)$, stage two the interpolation $f(z_1, z_2, y_{3,1}, \ldots)$ and so on. After the first stage, one interpolates each coefficient of the previous stage as a univariate polynomial. The main advantage of the Zippel algorithm is the following. If a polynomial coefficient c_α evaluates to zero at one stage, one assumes that it will also vanish at all other stages. This can avoid a sizable amount of black-box probes compared to a dense interpolator. Since this assumption is probabilistic, it is crucial to choose the numbers at which the polynomial is probed with great care in order for this assumption to hold. To make this choice more robust, one usually uses seed numbers which are called anchor points. The anchor points should be set to random numbers to minimize the chance of coincidental cancellations. All other z_i order values can be obtained by computing the corresponding power of the anchor point, i.e. if $y_{i,1} = 2$ then the corresponding jth z_i order value is $y_{i,j} = 2^j$.

For illustration we provide an example of the Zippel algorithm. Consider the following polynomial to be interpolated

$$f(z_1, z_2, z_3) = c_{\alpha_1} z_1^5 + c_{\alpha_2} z_1 z_2^4 + c_{\alpha_3} z_1 z_2 z_3^3 + c_{\alpha_4} z_2^5. \tag{9}$$

The indices α_i are the multi-index of the corresponding monomial, i.e. $\alpha_1 = (5, 0, 0)$. In the first stage, one interpolates the univariate polynomial in z_1 with Newton's algorithm by fixing $z_2 = y_{2,1}$ and $z_3 = y_{3,1}$. This yields

$$f(z_1, y_{2,1}, y_{3,1}) = k_0(y_{2,1}, y_{3,1}) + k_1(y_{2,1}, y_{3,1}) \cdot z_1 + k_5(y_{2,1}, y_{3,1}) \cdot z_1^5 \tag{10}$$

after six black-box probes. An additional probe is needed to verify the termination of this stage. The polynomial coefficients $k_i(y_{2,1}, y_{3,1})$ are multivariate polynomials in the remaining variables z_2 and z_3 evaluated at $y_{2,1}$ and $y_{3,1}$.

At stage two, the coefficients $k_i(y_{2,1}, y_{3,1})$ are promoted to polynomials $k_i(z_2, y_{3,1})$. Since $k_2 = k_3 = k_4 = 0$, we also assume the corresponding polynomials $k_i(z_2, z_3)$ to vanish. Therefore, only three polynomials have to be interpolated in z_2 at stage two. The univariate interpolation in z_2 is done by using the numerical values of $k_i(y_{2,j}, y_{3,1})$ of the polynomials

$$f_j(z_1, y_{2,j}, y_{3,1}) = k_0(y_{2,j}, y_{3,1}) + k_1(y_{2,j}, y_{3,1}) \cdot z_1 + k_5(y_{2,j}, y_{3,1}) \cdot z_1^5, \tag{11}$$

as black-box probe for the univariate interpolation of $k_i(z_2, y_{3,1})$. The result of the first stage, $f_1(z_1, y_{2,1}, y_{3,1})$ given by Eq. (10), can be reused. The other polynomials $f_j(z_1, y_{2,j}, y_{3,1})$ can again be interpolated by a univariate Newton interpolation. However, since we already

know that only three coefficients contribute to the univariate polynomial $f_j(z_1, y_{2,j}, y_{3,1})$, it is more efficient to build a system of three equations and solve it. Zippel's original algorithm requests new polynomials $f_j(z_1, y_{2,j}, y_{3,1})$ until all $k_i(z_2, y_{3,1})$ are interpolated, i.e. six times in the example. This means that stage two would require $6 \times 3 = 18$ black-box probes in total.

Kaltofen, Lee, and Lobo suggested to remove those $k_i(z_2, y_{3,1})$ from the system where the Newton interpolation terminates, which reduces the size of the system for the remaining coefficients [7, 8]. This procedure is called *temporary pruning*. In the example, the interpolation of $k_3(z_2, y_{3,1})$ terminates after the second step, the evaluation of $k_1(z_2, y_{3,1})$ after the sixth, and $k_0(z_2, y_{3,1})$ after the seventh, which corresponds to $3 + 4 \times 2 + 1 = 12$ black-box probes for stage two.

The result of stage two is

$$\begin{aligned} f(z_1, z_2, y_{3,1}) &= \tilde{k}_0(y_{3,1}) \cdot z_2^5 + (\tilde{k}_1(y_{3,1}) \cdot z_2 + \tilde{k}_2(y_{3,1}) \cdot z_2^4) z_1 + \tilde{k}_5(y_{3,1}) \cdot z_1^5 \\ &= \tilde{k}_0(y_{3,1}) \cdot z_2^5 + \tilde{k}_1(y_{3,1}) \cdot z_1 z_2 + \tilde{k}_2(y_{3,1}) \cdot z_1 z_2^4 + \tilde{k}_5(y_{3,1}) \cdot z_1^5. \end{aligned} \tag{12}$$

Stage three starts by promoting $\tilde{k}_i(y_{3,1})$ to polynomials $\tilde{k}_i(z_3)$ and proceeds exactly as stage two. The interpolations of $\tilde{k}_0(z_3)$, $\tilde{k}_2(z_3)$, and $\tilde{k}_5(z_3)$ terminate after the second step, and the interpolation of $\tilde{k}_1(z_3)$ after the fifth. Thus, Zippel's original algorithm requires $4 \times 4 = 16$ and the improved version $4 + 3 \times 1 = 7$ black-box probes for stage three.

Therefore, the complete interpolation of Eq. (9) requires $7 + 18 + 16 = 41$ black-box probes with the original version and $7 + 12 + 7 = 26$ with the improved version using temporary pruning. In contrast, the recursive multivariate Newton interpolation needs $(5 + 2) \times (5 + 2) \times (3 + 2) = 245$ probes.

The Zippel algorithm can be further improved by assuming knowledge of the maximal degree R, which always is the case when using it as part of the multivariate interpolation of rational functions as described in Sect. 2.2. Then, one can abort the univariate Newton interpolation of k_i after step j if $j = R - r_i$, where r_i is the total degree of the monomial corresponding to k_i. This procedure is called *permanent pruning* [57]. This reduces the total number of black-box probes to 20 for the given example, whereas the multivariate Newton algorithm only drops to $(5 + 1) \times (5 + 1) \times (3 + 2) = 180$ probes. In contrast, there are $\binom{3+5}{5} = 56$ possible coefficients and solving for them with a system of equations requires the same amount of black-box probes. This clearly shows the advantage of the Zippel algorithm compared to a recursive multivariate Newton interpolation.

Furthermore, if we know that all monomials have the same degree $r = 5$, the example already finishes after stage two. From Eq. (12) we can deduce that $c_{\alpha_3} = \tilde{k}_1(y_{3,1})/y_{3,1}^3$ is the coefficient for $z_1 z_2 z_3^3$. This prediction of the remaining powers is not used in our implementation of the Zippel algorithm, because it is closely related to the homogenization precedure [57] which will be done for the rational interpolation in Sect. 2.2.

Assuming a degree bound R, the Zippel algorithm scales as $O(nRt)$, with t being the number of non-zero coefficients [2]. In the completely dense case, the improved version with pruning requires $\binom{n+R}{R}$ probes with a degree bound, which is exactly one probe for every coefficient.

Without a degree bound, only a small number of additional probes is required to verify that the interpolation terminates.

An additional optimization regarding runtime can be achieved by noticing that the systems of equations which have to be solved during each stage in Zippel's algorithm are generalized transposed Vandermonde systems [3]

$$\begin{pmatrix} 1 & 1 & \dots & 1 \\ v_{\alpha_1} & v_{\alpha_2} & \dots & v_{\alpha_m} \\ \vdots & \vdots & \ddots & \vdots \\ v_{\alpha_1}^{m-1} & v_{\alpha_2}^{m-1} & \dots & v_{\alpha_m}^{m-1} \end{pmatrix} \begin{pmatrix} c_{\alpha_1} \\ c_{\alpha_2} \\ \vdots \\ c_{\alpha_m} \end{pmatrix} = \begin{pmatrix} f(\vec{y}^{\,0}) \\ f(\vec{y}^{\,1}) \\ \vdots \\ f(\vec{y}^{\,m-1}) \end{pmatrix}, \tag{13}$$

where $f(\vec{z})$ is a multivariate polynomial with m terms,

$$v_{\alpha_i} = \vec{y}^{\,\alpha_i} = \prod_{j=1}^{n} y_{j,1}^{\alpha_i[j]}, \qquad \vec{y}^{\,i} = \prod_{j=1}^{n} y_{j,1}^{i}. \tag{14}$$

All v_{α_i} have to be increasing monotonically with respect to α_i, with α_1 being the lowest degree. Solving algorithms for Vandermonde systems are much more efficient compared to Gaussian elimination, since Vandermonde systems can be solved using only $O(m^2)$ time and $O(m)$ space, where m is the number of equations [3, 5]. For usual Vandermonde systems presented in Eq. (13), the first evaluation of $f(\vec{z})$ will be done while setting all variables equal to 1. This variable choice increases the probability that monomials cancel, which is especially dangerous if $f(\vec{z})$ is the denominator of a rational function. To circumvent this problem, we propose to consider a modified version of generalized transposed Vandermonde systems:

$$\begin{pmatrix} v_{\alpha_1} & v_{\alpha_2} & \dots & v_{\alpha_m} \\ v_{\alpha_1}^2 & v_{\alpha_2}^2 & \dots & v_{\alpha_m}^2 \\ \vdots & \vdots & \ddots & \vdots \\ v_{\alpha_1}^m & v_{\alpha_2}^m & \dots & v_{\alpha_m}^m \end{pmatrix} \begin{pmatrix} c_{\alpha_1} \\ c_{\alpha_2} \\ \vdots \\ c_{\alpha_m} \end{pmatrix} = \begin{pmatrix} f(\vec{y}^{\,1}) \\ f(\vec{y}^{\,2}) \\ \vdots \\ f(\vec{y}^{\,m}) \end{pmatrix}. \tag{15}$$

These systems preserve all properties of general Vandermonde systems and only require a modification of the solving algorithm, while making cancellations very unlikely. An algorithm to solve such systems is given in App. A.

Note that it is beneficial in terms of the number of black-box probes to choose the variable order descending in the maximal degrees, i.e. that the lowest degree variable should be the last one to interpolate and the highest the first. This follows from the nature of the Zippel algorithm in which intermediate systems of equations usually grow with every stage. Generally, the best order for a black-box function is not known a priori, but for physical calculations one can utilize problems which are similar but simpler to obtain a guess on how the individual degrees could be distributed. We will illustrate the impact on the number of black-box probes of such a choice in Sect. 3.3.

Instead of using a univariate Newton interpolation in Zippel's algorithm for the multivariate polynomial interpolation, Refs. [7, 8] present a racing algorithm which races the sparse Ben-Or/Tiwari algorithm [4, 6] against the dense Newton interpolation. This procedure may significantly reduce the number of black-box probes by combining the advantages of both

a sparse and a dense algorithm. Further optimizations of the Ben-Or/Tiwari algorithm are described in Ref. [9]. We leave these optimizations for future versions.

2.2 Interpolation of rational functions

Rational functions can be constructed by combining two polynomials. Given two polynomials $P, Q \in \mathbb{Z}_p[\vec{z}]$, we define a rational function $f \in \mathbb{Z}_p(\vec{z})$, where $\mathbb{Z}_p(\vec{z})$ is the field of rational functions in the variables $\vec{z}$, as the ratio of P and Q:

$$f(\vec{z}) = \frac{P(\vec{z})}{Q(\vec{z})} = \frac{\sum_i n_{\alpha_i} \vec{z}^{\,\alpha_i}}{\sum_j d_{\beta_j} \vec{z}^{\,\beta_j}}. \tag{16}$$

The coefficients $n_{\alpha_i}, d_{\beta_j}$ are members of the field $\mathbb{Z}_p$ corresponding to multi-indices α_i and β_i.

Rational functions are not uniquely defined since their normalization is arbitrary. In order to provide a unique representation, we define the lowest degree coefficient in the denominator to be equal to one. If several monomials contribute to the lowest degree $d_{\min}$, we choose to define that coefficient of the monomial $\vec{z}^{\,\alpha}$ to be equal to one whose multi-index α is the smallest in a colexicographical ordering, e.g.

$$(1, 1, 0) < (1, 0, 1) < (0, 1, 1), \tag{17}$$

for $r = 2$.

The best strategy for interpolating rational functions is highly dependent on the available information. In the univariate case one can use Thiele's interpolation formula [1]. In the multivariate case, the best algorithm in terms of black-box probes is to solve a dense system of equations if one knows the terms of the rational function but not the values of the coefficients. Of course, this also works if one can restrict the number of terms by knowing bounds for the degrees of numerator and denominator. However, this is then a completely dense interpolation. In the early 1990s, Grigoriev, Karpinski, and Singer presented an algorithm to obtain bounds and then build a system of equations [10–12]. This approach can have a bad scaling behavior for many unknowns and high degrees since Gaussian elimination scales as $O(m^3)$ in time and $O(m^2)$ in space, with m being the number of equations.

In 2006, Khodadad and Monagan published an algorithm [13] which is based on the modified versions of the Extended Euclidean Algorithm [14, 18, 58]. The algorithm presented by Kaltofen and Zhang in 2007 [15] evaluates numerator and denominator separately [59, 60] and then uses a multivariate polynomial interpolation. It also exploits the sparsity of rational functions. Huang and Goa presented an algorithm in 2017 [17] based on Kronecker substitution [14]. These three algorithms all require bounds on the degrees of numerator and denominator.

In 2010, Cuyt and Lee presented an algorithm which uses the results of univariate interpolations with Thiele's interpolation formula as input for multivariate polynomial interpolations [16]. It is this algorithm which we adopt for `FireFly`, because it requires no information apart from the number of variables a priori and it utilizes the sparsity of a rational function.

Thus, we start by discussing how to interpolate a univariate rational function. According to Thiele [1] one can express a rational function $f \in \mathbb{Z}_p(t)$ as a continued fraction

$$T(t) = b_0 + (t - t_1)\left(b_1 + (t - t_2)\left(b_2 + (t - t_3)\left(\cdots + \frac{t - t_n}{b_n}\right)^{-1}\right)^{-1}\right)^{-1}, \tag{18}$$

with $t_1, \ldots, t_{n+1}$ being distinct elements of $\mathbb{Z}_p$. The coefficients $b_0 \ldots, b_n$ can be obtained recursively by numerical evaluations of the rational function f at $t_1, \ldots, t_{n+1}$,

$$b_i \equiv b_{i,i}, \tag{19}$$

$$b_{i,j} = \frac{t_{i+1} - t_j}{b_{i,j-1} - b_{j-1}}, \tag{20}$$

$$b_{i,0} = f(t_{i+1}). \tag{21}$$

The termination criterion is reached if one finds agreement between $f(t_i)$ and $T(t_i)$. Note that this approach only scales optimally with the number of black-box probes if the degree of the polynomials of numerator and denominator differ at most by one.

The idea of the algorithm of Ref. [16] is to perform the multivariate rational function interpolation with a dense univariate rational function interpolation and a sparse multivariate polynomial interpolation. First, we assume a rational function $f(\vec{z})$ of n variables which has a constant in the denominator and discuss the general case later. The constant is used to normalize the rational function and set to one. One starts by introducing a homogenization variable t and defining a new function $\tilde{f}(t\vec{z})$ as [57]

$$\tilde{f}(t\vec{z}) = f(tz_1, \ldots, tz_n). \tag{22}$$

We can interpret $\tilde{f}$ as a univariate rational function in the variable t, whose coefficients are multivariate polynomials in $\vec{z}$. Their degrees are bounded by the corresponding degree of t. Thus, by interpolating $\tilde{f}$ in t, we can use its coefficients for multivariate polynomial interpolation as described in Sect. 2.1. As optimization, we can set $z_1 = 1$ and reconstruct its power by homogenizing with respect to the corresponding power of t. To interpolate $\tilde{f}$, we generate random anchor points $y_{i,1}$ for the remaining z_i and perform a Thiele interpolation by evaluating $\tilde{f}(t\vec{y}^{\,1})$ for several random t until it terminates. We proceed for each variable sequentially ($z_1 \to z_2 \to \cdots \to z_n$) until the polynomial interpolation terminates. Ref. [16] uses the racing algorithm of Refs. [7,8] mentioned in Sect. 2.1 for the polynomial interpolation instead of Zippel's algorithm with a univariate Newton interpolation as we do.

Since Thiele's interpolation is not optimal for general rational functions in terms of the number of black-box probes, we use the knowledge gained during the first interpolation and subsequently only solve univariate systems of equations for all remaining coefficients. Additionally, we remove already solved coefficients from the system, to further reduce the number of black-box probes. The univariate system of equations can then easily be constructed by evaluating $\tilde{f}(t\vec{y})$ for different t at a fixed $\vec{z} = \vec{y}$ and construct each equation as

$$\sum_i n_{\mathrm{u},i}(\vec{y}) t^{r_i} - \tilde{f}(t, \vec{y}) \sum_j d_{\mathrm{u},j}(\vec{y}) t^{r_j} = \tilde{f}(t, \vec{y}) \sum_j d_{\mathrm{s},j}(\vec{y}) t^{r_j} - \sum_i n_{\mathrm{s},i}(\vec{y}) t^{r_i}, \tag{23}$$

where the subscripts s and u denote the solved and unsolved coefficients, respectively. In this notation the normalizing coefficient is included in the solved subset. Each of the solved coefficients can be computed by evaluating the corresponding multivariate polynomial at $\vec{z} = \vec{y}$. When enough equations have been obtained, the system can be solved for the unsolved coefficients. The time spent solving such univariate systems is usually negligible compared to the evaluation of the black-box function.

Until now, we have assumed that the denominator of f has a constant term. The generalization of the presented algorithm to arbitrary rational functions leads to a normalization problem. Assuming no constant term, we cannot set the coefficient of the the lowest degree monomial in the denominator of $\tilde{f}$ to one since this coefficient is a polynomial and not a constant. The algorithm presented in Ref. [16] cures this problem by introducing a variable shift $\vec{s} = (s_1, \ldots, s_n)$ such that $f(\vec{z}) \to f(\vec{z} + \vec{s}) \equiv \hat{f}(\vec{z})$ and $\hat{f}$ has a constant term in the denominator. A reliable way to fulfill this condition is to choose random but distinct values for all s_i. This shift comes with the caveat that $\hat{f}$ will be a much denser rational function in general. Especially, aside from accidental cancellations, it will contain all univariate terms in t up to the maximal degrees of numerator and denominator.

For simplicity we will focus on the numerator, noting that the following procedure can be applied completely analogously to the denominator. Assume that the maximal degree of the numerator of $\hat{f}$ in t is R with its corresponding coefficient n_R. Note that n_R will not be affected by the shift, thus, one starts by interpolating only this polynomial. The numerical values of n_i for $i < R$ are stored for later usage. Once n_R has been interpolated, we can remove it from the system of equations as before to reduce the number of black-box probes, and additionally subtract the effect of the shift on the lower degree coefficients to preserve their sparsity. The latter can be achieved by analytically evaluating

$$n_R(\vec{z} + \vec{s}) - n_R(\vec{z}). \tag{24}$$

It is convenient to save the results of Eq. (24) split according to their corresponding degree in t. Subsequently, one moves to n_{R-1} and subtracts the term of Eq. (24) corresponding to the degree $R-1$ evaluated at the same $y_{i,j}$ as the current interpolation point. This removes the shift from n_{R-1} and preserves its sparsity in the multivariate polynomial interpolation.

To illustrate the above algorithm, we present the following example which will be interpolated over $\mathbb{Z}_{509}$. Assume we want to interpolate the function

$$f(\vec{z}) = \frac{3z_1 + 7z_2}{z_1 + z_2 + 4z_1z_2}. \tag{25}$$

We start by choosing the anchor points $\vec{y}^{\,1}$ for $\vec{z}$, and homogenize with t:

$$y_{1,1} = 1, \qquad y_{2,1} = 10 \quad \Rightarrow \quad \tilde{f}\left(t\vec{y}^{\,1}\right) = \frac{73t}{11t + 40t^2}. \tag{26}$$

Since there is no constant term in the denominator, we perform a variable shift $\vec{s}$ to find a unique normalization:

$$s_1 = 4, \qquad s_2 = 1 \quad \Rightarrow \quad \hat{f}\left(t\vec{y}^{\,1}\right) = \frac{316 + 464t}{1 + 178t + 317t^2}. \tag{27}$$

The function $\hat{f}\left(t\vec{y}^{\,1}\right)$ in Eq. (27), whose coefficients are already uniquely normalized, is the result of the Thiele interpolation in the first step of the algorithm. Since we now know the maximal degrees of numerator and denominator, we can proceed in the remaining stages by solving systems of equations instead of using Eq. (18) to reduce the number of black-box probes. The numerical coefficients of $\hat{f}\left(t\vec{y}^{\,1}\right)$ are multivariate polynomials evaluated at $t\vec{y}^{\,1}+\vec{s}$ and, thus, are used as the input for the following polynomial interpolation according to Sect. 2.1. We will now focus on the numerator. The denominator can be interpolated in an analogous way. Assume we have fully interpolated the highest degree of the numerator as

$$P_1' = 291 + 170z_2, \tag{28}$$

which will yield

$$P_1 = 291z_1 + 170z_2 \tag{29}$$

after homogenization. The prime indicates the non-homogenized coefficient. To interpolate n_0 without the shift, we have to remove its effect before performing the polynomial interpolation. The input for the latter will thus be

$$316 - (P_1(\vec{z}+\vec{s}) - P_1(\vec{z})) = 316 - (146 + 170) = 0. \tag{30}$$

This is expected since there is no constant term in the numerator of $f(\vec{z})$. In total it takes twelve black-box probes to interpolate Eq. (25) using this algorithm. One can also apply this example to the case where $d_0 \neq 0$ with $\vec{s} = 0$. To further reduce the probes mandatory for this algorithm to succeed, the variable with the highest degree should be set to one and is thus only obtained by homogenization of the degrees after the polynomial interpolation finishes. This step requires neither additional black-box evaluations nor the use of interpolation algorithms.

Note that there is no need that the denominator comes with a constant. The same procedure can be applied if one finds a constant in the numerator after introducing a shift and using the latter for normalization. Additionally, one can apply algorithms to find a shift in fewer variables to reduce the higher complexity introduced by the normalization ambiguity. A straightforward algorithm is to test different shifts and apply the one, which shifts a minimal number of variables to obtain a unique normalization. To achieve this, in a first run the maximal degrees of numerator and denominator of the homogenized rational function have to be determined. Since we assume no knowledge about the black box, it is necessary to shift all variables to guarantee the existence of constants in numerator and denominator and thus prevent cancellations in the homogenization variable. Afterwards, one tests different shift configurations and checks if the maximal degrees of numerator and denominator coincide with the degrees obtained when shifting all variables. For practical usage, these steps may be important for sparse black-box functions with four or more variables since a shift including all arguments can lead to $O(\binom{n+R}{R})$ additional terms, where n is the number of variables and R the maximal degree of numerator or denominator, which have to be calculated to remove the effect of the shift in the interpolation of multivariate polynomials.

2.3 From finite fields to rational numbers

In the previous sections, we described how one can interpolate multivariate polynomials and rational functions over a finite field. Since their coefficients are only valid in the specific field used for the interpolation, one has to promote these numbers to rational numbers such that the corresponding functions are elements of $\mathbb{Q}[\vec{z}]$ and $\mathbb{Q}(\vec{z})$, respectively. Therefore, we need to reconstruct the interpolated results over the rational field. Generally, there is no inversion of the mapping from rational numbers to members of a finite field, but one can use a method called rational reconstruction (RR). This method is based on the Extended Euclidean Algorithm [14] and the first algorithm was described by Wang in 1981 [18], which is presented in Alg. (1).

This algorithm leads to a guess for a rational number $a = b/c$, with b and c being integers, from its image

$$e = a \bmod p \in \mathbb{Z}_p. \tag{31}$$

It will succeed if $|b|, |c| \leq \sqrt{p/2}$. In Ref. [61] it was proven that a is unique if it is found for a given e and p. However, this does not mean that the a is the correct rational number in $\mathbb{Q}$ which one desires to reconstruct, because the unique guess can be different in other prime fields. Also, the bound of $\sqrt{p/2}$ could lead to failures of the RR if p is restricted to machine-size integers.

Algorithm 1 Modified Euclidean Algorithm for rational reconstruction based on [18].

Input: A finite field member `a`, the defining prime of the finite field `p`.
Output: If succeeded a unique value of `a` in $\mathbb{Q}$ else an error.

```
function RATIONAL_RECONSTRUCTION(a, p)
    t ← 1;   old_t ← 1;
    r ← p;   old_r ← a;
    while 2 * r**2 > p do
        quotient ← ⌊ old_r / r ⌋;
        (old_r , r) ← (r, old_r - quotient * r);
        (old_t, t) ← (t, old_t - quotient * t);
    end while
    if 2 * t**2 > p  or  gcd(r, t) ≠ 1 then
        Throw an error that the rational reconstruction failed;
    end if
    return sign(t) * r / |t|;
end function
```

Both problems can be solved by the Chinese Remainder Theorem (CRT) [14], which comes with the cost of additional interpolations in other prime fields. The theorem states that one can uniquely reconstruct an element in $\mathbb{Z}_p$ from its images $\mathbb{Z}_{p_i}$ with $i = 1, \ldots, l$ by combining pairwise coprime numbers p_i to $p = p_1 \cdot p_2 \cdot \cdots \cdot p_l$ and applying the RR using p. An algorithmic realization is shown in Alg. (2). To reconstruct a rational number, one can then combine as many prime numbers as are required to successfully perform the RR. We then consider the rational number to probably be the correct number in $\mathbb{Q}$ if the RR yields the same result for two different fields. This guess will be tested later.

Algorithm 2 Chinese Remainder Theorem algorithm CRT [14].

Input: Two pairs containing a prime `p_i` of a finite field and a coefficient `c_i` obtained by the functional reconstruction over this field.

Output: A pair consisting of the combined value of the `c_i`'s and `p_i`'s according to the CRT.

```
function CHINESE_REMAINDER((c_1, p_1), (c_2, p_2))
    p_3 ← p_1 * p_2;
    m_1 ← (1 / p_2 % p_1) * p_2;
    m_2 ← (1 - m_1) % p_3;
    c_3 ← (m_1 * c_1 + m_2 * c_2) % p_3;
    return (c_3, p_3);
end function
```

Alg. (1) is not an optimal algorithm for arbitrary b and c because it will only succeed if both $|b|$ and $|c|$ are smaller than $\sqrt{p/2}$. Thus, in the worst case they differ by many orders in magnitude and only one of them fails the bound. In principle, it is possible to have different bounds for b and c. However, without additional knowledge one does not know a priori how to choose them. In Ref. [58] it was observed that the guess of the rational number comes together with a huge quotient in the Euclidean Algorithm. The suggested algorithm based on this observation is called Maximal Quotient Rational Reconstruction and presented in Alg. (4) in Sect. A. It can only be proven that it returns a unique solution if $|b||c| \leq \sqrt{p}/3$. However, it performs much better in the average case because large quotients from random input are rare. The parameter T allows to choose a lower bound for the quotients and, thus, the tolerance for false positive reconstructions. For `FireFly` we adjusted it to get roughly 1 % false positive results.

We then race Alg. (1) against Alg. (4) and consider a guess for a rational number as probably correct if either of the two algorithms reconstructs the same number in two consecutive fields. Both algorithms can return false positive results, i.e. unique rational numbers which are not the correct number in $\mathbb{Q}$. However, we consider the probability to reconstruct the same false positive number in two different fields as negligible. We also consider a guess as probably correct if it is an integer fulfilling the condition of Alg. (1) or if the combined integer does not change after applying the CRT.

Combining the RR, the CRT, and the interpolation of functions over finite fields, the generic algorithm to obtain an interpolated function in $\mathbb{Q}$ is shown in Alg. (3). The algorithm succeeds when the function build up by the probably correct coefficients in $\mathbb{Q}$ coincides with the black-box probe evaluated in a new prime field. We also take all coefficients into account were either Alg. (1) or Alg. (4) was able to reconstruct a guess, which does not fulfill the criteria mentioned aboved, because it still could be correct.

After the first prime field, one obtains full knowledge about the functional form and only the coefficients have to be promoted to $\mathbb{Q}$. Thus, one can use this knowledge and only reconstruct those coefficients which cannot be promoted to $\mathbb{Q}$ or are not valid in all used prime fields. This can be done most efficiently in numbers of black-box probes by solving systems of equations and remove all already known coefficients. Additionally, no variable

Algorithm 3 Functional reconstruction algorithm over $\mathbb{Q}$ based on [44].

Input: A sequence of primes $p_1, \ldots, p_l$ and a black-box function $f(\vec{z})$.
Output: The result of $f(\vec{z})$ in $\mathbb{Q}$ with high probability.

```
for i ← 1; i ≤ n do
    if i == 1 then
        Interpolate the function f over Z_{p_1}(z⃗) and store the result;
        Use the RR to promote the stored result to a guess g ∈ Q(z⃗);
    else
        if the previous rational reconstruction succeeded then
            Evaluate g over the new field Z_{p_i} for a given z⃗;
            if g(z⃗) == f(z⃗) then
                Terminate the algorithm and set g(z⃗) = f(z⃗);
            end if
        else
            Reconstruct the function f in Z_{p_i} and store the result;
            Use the CRT to combine the function in Z_{p_i} with the stored result in Z_{p_1···p_{i-1}}
            and store the combined result;
            Use the RR to promote the stored result in Z_{p_1···p_i} to a guess g ∈ Q(z⃗);
        end if
    end if
    i ← i + 1;
end for
```

shift is required to solve the normalization problem anymore.

However, the simple idea to construct a single system of equations for all coefficients which still need to be determined has a major drawback. It requires $O(m^3)$ operations to solve this system with a standard algorithm. The size of the system can become quite large if the function has many variables and is densely populated. Solving this system can then become much more expensive than just interpolating the function from scratch if the time cost of the black-box probes is not too high. For polynomials, however, this is can be directly solved by using a generalized transposed Vandermonde system.

To circumvent this problem for rational functions, we break the system of equations down to several subsystems. We first build a univariate system of equations in the homogenization variable for all degrees which contain unsolved coefficients. This system is in general much smaller than a complete multivariate system and, thus, cheap to solve. The result for each univariate coefficient is again a multivariate polynomial in all remaining variables and thus used as input for the corresponding multivariate Vandermonde system. As soon as we have enough values to solve one of these systems, we solve it and remove the univariate degree from the following univariate systems of equations which are needed to calculate the remaining coefficients. Therefore, the number of black-box probes is the same as for the full multivariate system of equations without homogenization.

However, a rational function has to be normalized to one of the coefficients in order to yield unambiguous results after solving the homogenized univariate system. Since the functional

form is known after the interpolation over the first prime field, we check all univariate degrees for the number of contributing monomials and if their coefficients have already been promoted to $\mathbb{Q}$. If there is only a single contributing monomial for any degree, we choose its coefficient as normalization by setting it to one. For illustration, let its univariate degree be r. Afterwards, we check again which monomial coefficients can be promoted to $\mathbb{Q}$ using this normalization. Therefore, the homogenized rational function evaluated at $\vec{z} = \vec{y}$ is given by

$$\tilde{f}(t,\vec{y}) = \frac{\vec{y}^{\,\alpha} t^r + \sum_i n_i(\vec{y}) t^{d_i}}{\sum_j d_j(\vec{y}) t^{d_j}} \qquad \text{or} \qquad \tilde{f}(t,\vec{y}) = \frac{\sum_i n_i(\vec{y}) t^{d_i}}{\vec{y}^{\,\alpha} t^r + \sum_j d_j(\vec{y}) t^{d_j}}, \tag{32}$$

where $n_i(\vec{y})$ and $d_j(\vec{y})$ are the coefficients of the univariate rational function evaluated at $\vec{z} = \vec{y}$, which are multivariate polynomials. The univariate system can then be build exactly as in Eq. (23) and its solutions for $n_{\mathrm{u},i}(\vec{y})$ and $d_{\mathrm{u},j}(\vec{y})$ serve again as input for the generalized Vandermonde systems as described in Sect. 2.2.

In the application to IBP reductions as described in Sect. 4 a normalizing coefficient can be found sometimes. This is due to the fact that the variables are the space-time dimension d and variables with a mass dimension. The rational functions occuring have a fixed mass dimension and, thus, numerator and denominator have fixed mass dimensions as well. Different powers of d then distribute the monomials to different univariate degrees. The known mass dimensions also make it possible to set one of the massive variables to one, because it can be unambiguously reconstructed after the calculation. This distributes the monomial degrees further.

The normalization can also be fixed if the coefficient of a univariate degree is completely known represented as multivariate polynomial, i.e. all coefficients of the polynomial are already promoted to $\mathbb{Q}$. The numerical value of the coefficient can then always be computed for all $\vec{z}$ and all prime fields, which allows us to use it as normalization.

Should there be no coefficent which can be used as normalization we reintroduce the shift so that we can normalize to the constant term. We then proceed as described above with the complication that we have to start with the highest degree and then remove the shift from lower degrees recursively as described in Sect. 2.2.

3 `FireFly`

We implemented the functional interpolation and rational number reconstruction algorithms described in the previous section in the `C++` library `FireFly`, which is publicly available at

`https://gitlab.com/firefly-library/firefly`

It requires

- A `C++` compiler supporting `C++11`
- `CMake` ≥ 3.1

- `FLINT` $\geq$ 2.5 (optional)
- `GMP` $\geq$ 6.1

The dependency on the GNU Multiple Precision Arithmetic Library (GMP) [62] is needed to utilize the CRT if a single prime field is not sufficient to reconstruct the black-box function.

After downloading the library, `FireFly` can be configured and compiled by running

```
cd $FIREFLY_PATH
mkdir build
cd build
cmake -DWITH_FLINT=true .. # Without FLINT: -DWITH_FLINT=false
make
```

where `$FIREFLY_PATH` is the path to the `FireFly` directory. When the compilation has finished, the build directory will contain the static and shared libraries `libfirefly.a` and `libfirefly.so`, respectively. Calling

```
make install
```

will additionally copy the corresponding header files and libraries to a specified directory which is by default the system `include` and `lib` directory, respectively. The prefix of both installation directories can be set with

```
cmake -DCMAKE_INSTALL_PREFIX=$PREFIX ..
```

where `$PREFIX` is the desired path prefix. The include and library files will then be installed to `$PREFIX/include` and `$PREFIX/lib`, respectively.

When using `FLINT` [63,64] additional flags have to be set if `FLINT`s header and library files cannot be found in the system `include` and `lib` directories. The configuration has thus to be provided with the absolute paths to the include directory and the shared library of `FLINT`. This can be done calling

```
cmake -DWITH_FLINT=true -DFLINT_INCLUDE_DIR=$FLINT_INC_PATH \
  -DFLINT_LIBRARY=$FLINT_LIB_PATH
```

where `$FLINT_INC_PATH` and `$FLINT_LIB_PATH` are the absolute paths to the include directory to the shared library of `FLINT`, respectively.

In the following sections, we describe the library `FireFly` and give examples of its usage.

3.1 Finite field integer and modular arithmetic

Most of the numerical computations are carried out over a finite field modulo a prime p. Since there is no built-in C++ implementation of such an object, we provide the class FFInt which denotes a finite field integer up to 64 bit. An FFInt holds basically three uint64_t objects: n which is the element of the field, p to set the defining prime and p_inv which is the inverse of p needed for modular arithmetic when using FLINT. To enable parallel interpolations of various functions over the same field and to save memory, the latter two are static variables. All basic arithmetic operations

+, -, /, *, +=, -=, *=, /=,

relational operators

==, !=, >, <, >=, <=,

and unary operators

+, -, !, ++, --

are implemented. The default implementation of modular arithmetic is given as an interface to routines of FLINT. Using the latter is optional but without this library, one has to provide own routines for all arithmetic operators and the pow member function which is mentioned in the next paragraph.

All public member functions and variables are summarized in the following:

`static void set_new_prime(uint64_t prime)`
: Static function that sets p = prime and calculates p_inv. The latter is only set if one compiles FireFly with FLINT.

`FFInt(const T n)`
: Creates an FFInt object by setting its value to n mod p where the template T can be of type FFInt, mpz_class, uint64_t, std::string, or any other primitive integer type.

`FFInt pow(const FFInt& e)`
: Returns a^e mod p.

`FFInt pow(const FFInt& a, const FFInt& e)`
: Returns a^e mod p.

`static uint64_t p`
: The defining prime p of the field.

`uint64_t n`
: The element n of the prime field $\mathbb{Z}_p$.

set_new_prime has always to be called first before performing modular arithmetic to define a prime field. In the following code snippet, we show the exemplary usage of the FFInt class:

```
FFInt::set_new_prime(509);
FFInt a(10);
FFInt b(13);
FFInt c(3);
FFInt d = pow(c, a); // d is now 5
FFInt e = a.pow(b) + c/a; // e is now 355
```

All member functions are understood to be part of the `firefly` namespace. `FireFly` provides the user with an array of the 100 largest 63 bit primes which can be accessed through the function `primes`. To get the largest 63 bit prime, one calls `primes()[0]` and has to include the header `ReconstHelper.hpp`. Note that we are only using `C++` STL containers apart from our custom container classes.

3.2 Reconstructing functions with `FireFly`

`FireFly` offers the interface class `Reconstructor` for the reconstruction of rational functions and polynomials. It is provided with a thread pool[1] to allow for the parallel interpolation of various black-box functions over the same prime field and the promotion of their coefficients to $\mathbb{Q}$. Its member functions can be summarized as:

`Reconstructor(uint32_t n, uint32_t thr_n, uint32_t verbosity = IMPORTANT)`
: Creates a `Reconstructor` object with `n` variables and `thr_n` threads. The verbosity levels are `SILENT`, `IMPORTANT`, and `CHATTY`.

`void enable_scan()`
: Enables the scan for a sparse shift at the beginning of the reconstruction. As `FireFly` assumes a variable ordering by degree, i.e. the first variable has the highest degree, the possible shifts are scanned starting by shifting only the last variable and proceeding variable by variable until the first. After this two variables are scanned starting again with the last two proceeding to the first two etc.

`void reconstruct()`
: Performs the reconstruction.

`vector<RationalFunction> get_result()`
: Returns the reconstructed functions as `RationalFunction` objects (c.f. App. B).

`void black_box(vector<FFInt>& result, const vector<FFInt>& values)`
: The user-provided black-box function, which fills the probes for the variable values `values` into `result`. The vector `result` is required to have immutable ordering. `values` contains n entries corresponding to n variables.

`void set_tags(const vector<string>& tags)`
: Enables storing of intermediate results in the directory `./ff_save`. Each function state is stored under the name given by the corresponding entry in `tags`. The same immutable ordering as for the probes in `black_box` is assumed.

[1] We adapted the thread pool used in `FlexibleSUSY 2.0` [65].

`void set_tags()`
: Same as above but set generic tag numbers starting at `0`.

`void resume_from_saved_state(const vector<string>& file_paths)`
: Resumes the reconstruction with the saved files given at the absolute paths `file_paths`.

To perform functional reconstructions, the user has to implement the function `black_box` which is provided with an empty result vector and a tuple of values at which the black box should be evaluated. The result vector has then to be filled by calling user defined functions to probe the black box. The `Reconstructor` class demands an immutable ordering of the result vector to verify the consistency of the reconstruction.

Below we show an example how one could use the `black_box` member function with dummy black boxes implemented in `FireFly`.

```
// Example of how one can use the black_box function of the interface
void Reconstructor::black_box(vector<FFInt>& result, const vector<FFInt>& \
  values) {
  result.emplace_back(singular_solver(values));
  result.emplace_back(n_eq_1(values[0]));
  result.emplace_back(n_eq_4(values));
  result.emplace_back(gghh(values));
  result.emplace_back(pol_n_eq_3(values));
  result.emplace_back(ggh(values));
}
```

To run the reconstruction for the above definded black-box function, the `Reconstructor` class could be used as follows:

```
// Construct the Reconstructor class with 4 variables and 4 threads
Reconstructor reconst(4, 4);
reconst.enable_scan(); // Enable the scan for a sparse shift
vector<string> tags = {"sing","n1","n4","gghh","pol","ggh"};
reconst.set_tags(tags); // Set tags to save states after each prime field
reconst.reconstruct(); // Reconstruct the functions
vector<RationalFunction> results = reconst.get_result(); // Get results
```

We first initialize the `Reconstructor` with four variables and four threads. Then, we enable the scan for a sparse shift and set tags to store intermediate results. `reconstruct` first scans the functions to check if a shift in a subset of the four variables is sufficient to always produce a constant term for the normalization and then performs the interpolation, manages the promotion to new prime fields and tries to reconstruct each coefficient over $\mathbb{Q}$. To parallelize the reconstruction, each probe is evaluated in its own thread and each black-box function is reconstructed by a `RatReconst` object (c.f. App. B) which itself performs the reconstruction single threaded.

Should the interpolation be aborted it can be resumed with

```
// Give the absolute paths to the intermediate results
vector<string> file_paths = {"ff_save/sing_3.txt", "ff_save/n1_2.txt", \
  "ff_save/n4_3.txt", "ff_save/gghh_4.txt", "ff_save/pol_1.txt", \
  "ff_save/ggh_2.txt"};
// Enables to resume from a saved state
reconst.resume_from_saved_state(file_paths);
reconst.reconstruct();
```

The file names consist of the tag given above and the last prime counter for which the object was saved. These counters can differ between the tags because the corresponding reconstruction can finish at different counters and already reconstructed functions will not produce new intermediate states to save.

If the black-box function requires the numerical result to be in a different format, e.g. a matrix in the form of a nested vector instead of a vector, the user has two options. One could either parse the result to a vector or implement the `Reconstruct` object oneself and adjust a few lines of code. We follow the latter approach for our application to IBP reductions in Sect. 4. More information about `FireFly` can be found in App. B.

3.3 Benchmarks

In this section we highlight the influence of different options which can strongly affect the time spent on a reconstruction and the corresponding memory consumption. For that purpose, we define the following benchmark functions:

$$f_1(z_1,\ldots,z_{20}) = \frac{\sum_{i=1}^{20} z_i^{20}}{\sum_{i=1}^{5} \left(z_1 z_2 + z_3 z_4 + z_5 z_6\right)^i z_{20}^{35}} , \tag{33}$$

$$f_2(z_1,\ldots,z_5) = \frac{123456789109898799879870980 \left(\left(1+\sum_{i=1}^{5} z_i\right)^{20} - 1\right)}{z_4 - z_2 + z_1^{10} z_2^{10} z_3^{10} z_4^{10} z_5^{10}} , \tag{34}$$

$$f_3(z_1,\ldots,z_5) = \frac{z_1^{100} + z_2^{200} + z_3^{300}}{z_1 z_2 z_3 z_4 z_5 + z_1^4 z_2^4 z_3^4 z_4^4 z_5^4} . \tag{35}$$

We test a sparse function with 20 variables (f_1), a dense function with five variables (f_2) which requires the usage of the CRT, and a sparse function with five variables and high individual degrees (f_3). All functions need a shift in at least one variable.

For the benchmarks we measure the total time, the number of black-box probes, and the total memory consumption using a single threaded setup without a thread pool. We compare different variable orderings and the influence of the scan for a sparser shift to the default options. All calculations are done on an Intel Core i7-3770 and 8 GiB of RAM. `FLINT` is enabled for modular arithmetic. The results of the benchmark tests are shown in Tab. 1 using different kinds of possible optimizations. To reconstruct f_2 we use the first five entries of the `primes` vector which can be found in the `ReconstHelper.hpp` file.

To fully reconstruct f_1, `FireFly` needs $\sim$ 90000 black-box probes using a parameter shift in all variables and a non-optimal ordering. This can be reduced to $\sim$ 20000 probes by using

Table 1: Benchmarks for f_1, f_2 and f_3 defined in Eqs. (33)-(35) obtained with `FireFly` using different optimizations.

Function	Shift scan	Variable order	Probes	Runtime	Memory usage
f_1	✗	$(z_1, \ldots, z_{20})$	87137	3.6 s	27.9 MiB
f_1	✗	$(z_{20}, z_1, \ldots, z_{19})$	41628	2.1 s	25.4 MiB
f_1	✓	$(z_1, \ldots, z_{20})$	84569	4.0 s	132.1 MiB
f_1	✓	$(z_{20}, z_1, \ldots, z_{19})$	22617	1.7 s	127.3 MiB
f_2	✗	$(z_1, \ldots, z_5)$	332893	3 min 8 s	54.9 MiB
f_2	✓	$(z_1, \ldots, z_5)$	320800	2 min 40 s	43.2 MiB
f_3	✗	$(z_1, \ldots, z_5)$	139710	42.4 s	13.7 MiB
f_3	✗	$(z_3, z_2, z_1, z_4, z_5)$	54211	9.5 s	9.4 MiB
f_3	✓	$(z_1, \ldots, z_5)$	137493	38.0 s	13.5 MiB
f_3	✓	$(z_3, z_2, z_1, z_4, z_5)$	34349	2.8 s	7.7 MiB

an optimal ordering and scanning for a sparse shift. The ordering alone leads to a reduction of $\sim$ 40000 probes, the search for a sparse shift requests 184 black-box evaluations but additionally reduces the total number of probes by $\sim$ 20000 leading to a reduced runtime of more than 50% from 3.6 s to 1.7 s taking both optimizations into account. Note that the memory consumption drastically increases when performing a shift scan for f_1 since we first calculate all possible permutations of the shift and store them in a vector. For 20 variables, this leads to $O(10^6)$ entries occupying $\sim$ 120 MiB of memory. The memory consumption of the reconstruction itself only needs $\sim$ 7 MiB and is thus reduced by roughly a factor of 4. The generation of the permuted shift vector requires 1.3 s whereas the reconstruction only takes 0.4 s. f_1 is constructed in a way that the ordering has a huge impact on the number of black-box probes required to reconstruct it. Thus, scanning only for a sparse shift and not reordering parameters leads to a shift in z_{20} which is the worst shift one could possibly choose. Therefore, almost non probes are avoided compared to shifting all variables and using the same variable ordering.

The optimization effects seen for f_1 are not that drastic when reconstructing a dense function like f_2 in which a parameter reordering is not helpful. Instead, only a scan for a sparser shift can avoid some probes and reduces the overall runtime in operations in which the shift needs to be removed. Thus, there is no significant benefit concerning memory consumption but a slightly faster runtime. Separated to different prime fields, the number of requested probes using a shift scan are: 204 to find a suitable shift, 213130 to interpolate the full functional dependence, 53130 for the second prime field since we can normalize to the univariate degree 50 term in the denominator, 53129 for the third prime field, 1206 for the fourth, and an additional probe to check that the reconstruction succeeded.

As for the reconstruction of f_1, one can observe similar benefits utilizing optimizations

for f_3. A reordering of variables can significantly reduce the number of black-box probes to approximately one third of the non-optimal ordered case and, additionally, shorten the runtime to almost one fifth. Scanning for a sparser shift without a reordering leads, by construction, only to a small number of black-box evaluations avoided compared to shifting all variables. The benefit of this option is reflected in the runtime which reduces by almost 4 s due to the much simpler polynomials needed to remove the effects of the shift during the polynomial interpolation. Combining an optimal variable order and a sparse shift can further reduce the runtime to 2.8 s while requiring only less than a quarter of the black-box probes needed using no optimizations.

Usually, there is no knowledge about the function to be interpolated. Therefore, a useful a priori assumption about the variable ordering cannot be made in general. However, for physical calculations it can be often useful to first study a similar but simpler problem to estimate the variable structure and apply a corresponding ordering to the actual calculation. It is always best to set the potentially highest degree variable to z_1 which will thus not be interpolated directly.

If evaluating probes is time consuming or the black-box function only depends on two to three variables, the scan for a sparse shift can lead to an increased overall runtime and might not be beneficial anymore.

4 Application to IBP reductions

In multi-loop calculations in high-energy physics one ends up with scalar Feynman integrals after an appropriate tensor reduction. The general form of scalar Feynman integrals with L loops is given by

$$I(d, \{p_j\}, \{m_i\}, \{a_i\}) \equiv \int_{k_1,\ldots,k_L} \frac{1}{P_1^{a_1} \cdots P_N^{a_N}} \tag{36}$$

with

$$\int_k \equiv \int \frac{\mathrm{d}^d k}{(2\pi)^d}. \tag{37}$$

and the inverse propagators $P_i = q_i^2 - m_i^2 + \mathrm{i}\epsilon$ in Minkowski space or $P_i = q_i^2 + m_i^2$ in Euclidean 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. N can be computed from L and the number of external momenta E by

$$N = EL + \frac{L(L+1)}{2}. \tag{38}$$

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

$$r \equiv \sum_{i=1}^{\#\text{prop.}} \theta\left(a_i - \frac{1}{2}\right) a_i \tag{39}$$

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

$$s \equiv \sum_{i=1}^{\#\text{prop.}} \theta\left(\frac{1}{2} - a_i\right) |a_i|, \tag{40}$$

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.

The integration-by-parts (IBP) algorithm of Chetyrkin and Tkachov [19, 20] is based on the observation that inserting the scalar product of a derivative with respect to a loop momentum with another momentum into Eq. (36) leads to a vanishing integral in dimensional regularization:

$$\int_{k_1,\ldots k_L} \frac{\partial}{\partial k_i^\mu} \left(\tilde{q}_j^\mu \frac{1}{P_1^{a_1} \ldots P_N^{a_N}} \right) = 0, \tag{41}$$

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

$$0 = \sum_n c_n I(d, \{p_j\}, \{m_i\}, \{a_i^{(n)}\}) \tag{42}$$

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

They can be combined to recursion relations which express an integral through easier integrals. The recursive application of the relations then allows to reduce all integrals to a small set of master integrals, which have to be computed by other methods to get the final result. At three-loop level, this approach has been successfully applied for massless propagator-type diagrams [20–22], massive tadpoles [23, 24], and on-shell propagators [25–27]. However, the recursion relations usually have to be derived manually which makes this procedure unfeasible for multi-loop calculations. On the other hand, the reduction is straightforward and fast should they be found. In the last decade, some progress in the automatic derivation of recursion relations has been made [28–31].

In 2001, Laporta presented a different strategy [32]. By inserting integer values for the a_i in Eq. (42), so-called *seeds*, one obtains a system of equations for the occuring integrals. These systems can be solved by standard procedures to yield the desired reduction to master integrals. Obviously, one has to choose a sufficient amount of seeds for the specific problem. There exist several public implementations of modified versions of the Laporta algorithm, `AIR` [33], `FIRE` [29, 34–36], `Reduze` [37, 38], and `Kira` [39, 40], and numerous private ones. However, the Laporta algorithm has some major drawbacks. The systems of equations for the state-of-the-art calculations become huge and expensive to solve both in terms of memory and runtime. This is partially related to large intermediate expressions which occur during the solution process.

Since the IBP relations are linear, the solution strategies only involve the addition, subtraction, multiplication, and division of the polynomial coefficients c_n. Therefore, the coefficients

of the master integrals are rational functions in d, $\{m_i\}$, and $\{p_j\}$. Thus, the problems of the Laporta algorithm can be eased by finite-field techniques as proposed by Kauers in 2008 [41]. One can replace all occuring variables by members of the finite field and the solve the system of equations numerically. This is in general orders of magnitude faster than solving the system analytically. The first realized application was to solve the system of equations over a finite field before the actual analytic reduction [42]. This allows to identify and remove the linearly dependent equations. By also performing the back substitution, one can additionally identify the master integrals and also select only those equations which suffice to reduce a requested subset of integrals. This procedure has been implemented in `Kira` [39].

Some of the finite-field-interpolation techniques from computer science described in Sect. 2 have been summarized before in a physical context in Refs. [43, 44]. The first pure finite-field IBP reduction was presented by von Manteuffel and Schabinger in 2016 [45]. Recently, three more calculations were concluded [46–48]. All of them are one scale problems and, thus, one variable problems, i.e. they only require univariate interpolation techniques, which has been implemented in private codes. The recently published Version 1.2 of `Kira` uses a multivariate polynomial interpolation at intermediate stages of the reduction if it seems useful [40]. This means that instead of multiplying some (sub-)coefficients algebraically, their numerical values are multiplied and the result is interpolated by Newton's interpolation formula. However, this interpolation is done over $\mathbb{Z}$ and not in a finite field.

Recently, `FIRE6` was published as first public implementation of the Laporta algorithm using the interpolation of rational functions over a finite field [36]. For polynomials it performs a multivariate Newton interpolation. The interpolation of rational functions works in several steps. First, univariate interpolations are performed for each variable separately using Thiele's interpolation formula. By assuming that the denominator of the final result can be factorized into functions depending only on one variable each, one can turn the rational function into a polynomial by multiplying it with the denominators obtained from the univariate interpolation. This allows to perform a multivariate Newton interpolation. However, it seems to only work for up to two variables at the moment and it is not clear whether the factorization assumption can be generalized to several kinematic variables.

For more details on IBP reductions we refer the reader to the Ref. [66] and Refs. [67,68] for an general overview about techniques for the calculation of multi-loop Feynman integrals.

4.1 Implementation

Since `Kira` already utilizes the reduction over a finite field as a preliminary step, we chose it as a basis for our implementation. The idea is to use the built-in solver `pyRed` to solve the system of IBPs many times over several prime fields, extract the numerical values, and feed them to `FireFly` to reconstruct the rational functions. We thus added the option to start the reduction with an interpolation in finite fields using `FireFly`. Another important option is to choose the order of the variables of the problem including d. The order can have a major impact on the runtime as shown in Sect. 3.3.

Our implementation follows the `Reconstructor` class described in Sect. 3.2 with several

small modifications, e.g. the structure of the function `black_box`. `FireFly` is compiled to use `FLINT` for the modular arithmetic. We start by loading the system of linearly-independent equations. The variables are replaced by the values requested by the reconstruction objects and the resulting numerical system is solved by `pyRed`. Additionally, we perform two integral selections. Even though the system only contains all required linearly-independent equations from the start, after the forward eliminations not all of these equations are required anymore for the back substitution in general. We thus select only those equations which are required for the reduction of the requested integrals. The same selection is performed in the analytic calculation with `Kira`. We perform a second selection after the back substitution which selects only the coefficients of the master integrals for the requested integrals. This selection does not offer any advantage in an analytic approach because everything is already solved at this point. In our approach this allows us to omit potentially difficult rational functions which are not required to get the desired result.

4.2 Examples

To illustrate the validity of the reconstruction approach for Laporta's algorithm for multi-scale Feynman integrals, we apply our private implementation described in the previous section to some example topologies. We compare it with the highly optimized algebraic program `Kira 1.2`.[2] It is important to stress that our link of `FireFly` to `Kira` has room for optimizations and that we leave real benchmarks for an official implementation. Therefore, we just compare the straight on calculations, i.e. we do not use some of the advanced features of `Kira` like the sectorwise forward elimination or the back substitution for subsets of master integrals. We also do not use the option `algebraic_reconstruction`, i.e. we compare a fully algebraic calculation to the finite field approach. In both approaches, we restrict r and s of the seed integrals by $r_{\max}$ and $s_{\max}$ and select the integrals with the option `select_mandatory_recursively` in the same range.

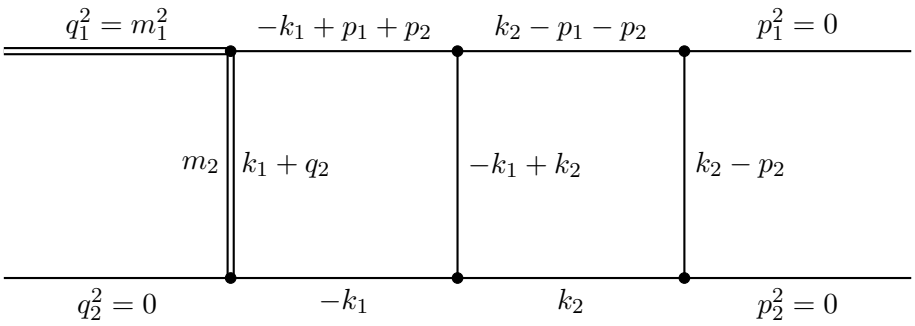

Figure 1: The planar double box `topo7` which occurs, e.g., in single top production.

Our first example is `topo7`, which is one of the examples provided by `Kira`. This planar two-loop diagram is shown in Fig. 1. It occurs in single top production. We first set the masses m_1 and m_2 to zero so that the kinematic invariants s[3] and t are the only massive parameters.

[2] We use it in combination with version 6.21 of `Fermat` [69].
[3] Not to be confused with the sum of all negative powers s defined in Eq. (40).

By setting s to one, only t and the space-time dimension d remain as parameters. For our reconstruction, we order them as d, t for the best performance, because d appears with higher powers in the result.

Table 2: Runtime and memory usage for `Kira 1.2` and `Kira` with `FireFly` for `topo7` without masses and with $r_{\max} = 7$. We use the option `select_mandatory_recursively`, order the variables as d, s, t, and set s to one. The runtime only includes the time spent on the forward elimination and back substitution or the time for the reconstruction, respectively. These tests were run on a computer with an Intel Core i7-3770 and 8 GiB of RAM.

	`Kira 1.2`		`Kira` with `FireFly`		
$s_{\max}$	Runtime	Memory usage	Runtime	CPU time for `pyRed`	Memory usage
2	7.9 s	0.3 GiB	1.9 s	89 %	0.3 GiB
3	10.2 s	0.5 GiB	5.2 s	90 %	0.5 GiB
4	14.1 s	0.9 GiB	14.3 s	90 %	0.9 GiB

The results for different $s_{\max}$ are shown in Tab. 2. For $s_{\max} = 2$, the reconstruction is faster by a factor of four. However, it scales worse than the algebraic calculation when increasing $s_{\max}$ with only eight threads available. `Kira` becomes slightly faster for the amplitude with $s_{\max} = 4$. `Kira` has to reduce 2774, 7994, and 17548 equations for $s_{\max} = 2, 3, 4$, whereas the reconstruction implementation selects only the 730, 1837, and 3970 mandatory equations, respectively. Most of the CPU time for the reconstruction is used for the numerical solutions of the system with `pyRed`, i.e. the black-box probes. For $s_{\max} = 2$, the reconstruction requires about 120 black-box probes which take 0.07 s, 180 with 0.17 s each for $s_{\max} = 3$, and 280 with 0.33 s each for $s_{\max} = 4$. The forward elimination contributes with roughly 75 % to these times. The memory consumption for both versions is dominated by the preliminary steps to construct the system of linearly independent equations.

The calculation becomes much more expensive in terms of CPU time when taking the masses into account. We again order the variables by the highest powers appearing in the result, which now is m_2, t, s, d, m_1 and set the highest massive parameter m_2 to one.

The two additional parameters lead to a drastic increase in the runtime up to two orders of magnitude for both approaches for higher $s_{\max}$, but the reconstruction approach suffers much more than the algebraic one as shown in Tab. 3. However, now the scaling with $s_{\max}$ is inverted. The reconstruction approach is slower by a factor of 3.4 for $s_{\max} = 2$, which reduces to a factor of 3.0 for $s_{\max} = 4$. The memory consumption is almost entirely dominated by the actual reduction or reconstruction with $s_{\max} = 2$ for the reconstruction being the sole exception. `Kira` reduces 3902, 10216, and 22571 equations for $s_{\max} = 2, 3, 4$, whereas our reconstruction implementation selects only the 949, 2439, and 5370 mandatory equations. The number of coefficients which have to be reconstructed increases from 2984 to 7428 to 15531. Since the complexity of the coefficients also increases when going to higher $s_{\max}$, they require a lot more memory to store input values, terms, and subtraction terms of the shift. The higher complexity also leads to a decreasing limitation through `pyRed`, because internal calculations in `FireFly` become more expensive. However, the numerical

Table 3: Runtime and memory usage for `Kira 1.2` and `Kira` with `FireFly` for `topo7` with $r_{\max} = 7$. We use the option `select_mandatory_recursively`, order the variables as m_2, t, s, d, m_1, and set m_2 to one. The runtime only includes the time spent on the forward elimination and back substitution or the time for the reconstruction, respectively. These tests were run on a computer with an Intel Core i7-3770 and 8 GiB of RAM.

	`Kira 1.2`		`Kira` with `FireFly`		
$s_{\max}$	Runtime	Memory usage	Runtime	CPU time for `pyRed`	Memory usage
2	38 s	0.5 GiB	128 s	94 %	0.4 GiB
3	270 s	0.8 GiB	880 s	91 %	0.8 GiB
4	3000 s	1.6 GiB	9200 s	89 %	3.6 GiB

solutions with `pyRed` are still the main bottleneck by an order of magnitude. Roughly 9400 of them are required for $s_{\max} = 2$, with each of them taking taking 0.1 s. These numbers increase to 25900 and 0.24 s for $s_{\max} = 3$ and 117000 and 0.54 s for $s_{\max} = 4$. Again, the forward elimination is responsible for more than 75 % of the solution times.

Table 4: Scaling of `Kira 1.2` and `Kira` with `FireFly` with the number of threads for `topo7` with $r_{\max} = 7$ and $s_{\max} = 4$. We use the option `select_mandatory_recursively`, order the variables as m_2, t, s, d, m_1, and set m_2 to one. The runtime only includes the time spent on the forward elimination and back substitution or the time for the reconstruction, respectively. These tests were run on a computer with two Intel Xeon Gold 6138 and 768 GiB of RAM.

	`Kira 1.2`		`Kira` with `FireFly`	
# Threads	Runtime	Memory usage	Runtime	Memory usage
10	1900 s	1.9 GiB	7450 s	3.6 GiB
20	1500 s	2.8 GiB	4350 s	3.9 GiB
40	1550 s	4.8 GiB	2950 s	4.1 GiB
80	1450 s	8.8 GiB	2050 s	4.9 GiB

In Tab. 4 we show how both approaches scale with the number of CPU threads available. Note that these numbers are not directly comparable to the numbers in Tab. 3, because different computers were used. The CPUs not only differ in clock speed and architecture but also in the number of cores. The Intel Core i7-3770 used in Tab. 3 has four physical cores and four additional logical cores whereas the two Intel Xeon Gold 6138 used for Tab. 4 offer 20 physical and 20 additional logical cores each. Therefore, the calculations in Tab. 4 can run on physical cores for up to 40 threads.

`Kira` does not really profit from many threads in this example. Doubling the number from 10 to 20 only increases the performance by less than a factor of 1.3. A further increase basically does not impact the runtime anymore, since large parts of the back substitution run on one or two threads. However, the memory consumption increases drastically, because

each additional `Fermat` instance requires additional memory. On the other hand, doubling the number of threads from 10 to 20 increases the performance of the reconstruction approach by a factor of 1.7, doubling from 20 to 40 by a factor of 1.5, and doubling from 40 to 80 still by a factor of 1.4. Thus, while the reconstruction takes almost four times longer than the algebraic approach on 10 threads, this difference reduces to only 40 % on 80 threads. Additionally, the reconstruction approach could profit from even more threads. The performance gain only comes with a moderate increase in memory consumption.

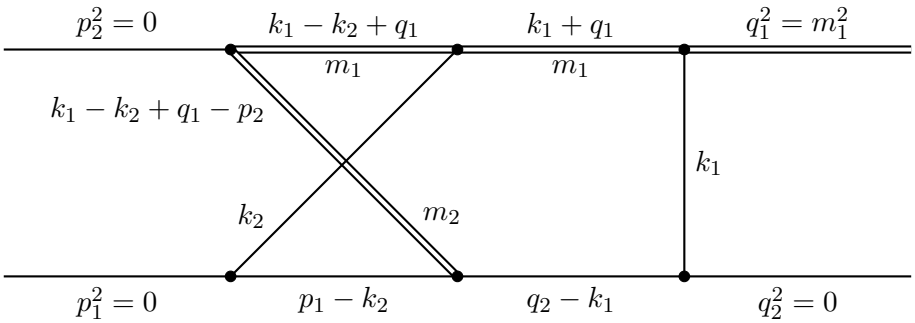

Figure 2: The non-planar double box `topo5` which occurs, e.g., in single top production.

Our second example is `topo5` of the `Kira` examples, which is a two-loop non-planar diagram occurring in single top production. The diagram is shown in Fig. 2. We order the parameters as m_1, s, m_2, t, d and set m_1 to one.

Table 5: Runtime and memory usage for `Kira 1.2` and `Kira` with `FireFly` for `topo5` with $r_{\max} = 7$. We use the option `select_mandatory_recursively`, order the variables as m_1, s, m_2, t, d, and set m_1 to one. The runtime only includes the time spent on the forward elimination and back substitution or the time for the reconstruction, respectively. These tests were run on a computer with two Intel Xeon Gold 6138 and 768 GiB of RAM.

	`Kira 1.2`		`Kira` with `FireFly`		
$s_{\max}$	Runtime	Memory usage	Runtime	CPU time for `pyRed`	Memory usage
1	4 min	7.6 GiB	3 min	99 %	0.9 GiB
2	1 h 53 min	33 GiB	1 h 42 min	97 %	3.3 GiB
3	18 h 28 min	102 GiB	18 h 34 min	91 %	18 GiB

Both versions perform the reductions in similar times as shown in Tab. 5. However, the reconstruction approach requires significantly less memory. Again, the CPU time is mainly used for `pyRed`. For $s_{\max} = 1, 2, 3$, the reconstruction requires 15500 probes and one prime field, 269700 probes and two prime fields, 1090400 probes and three prime fields. Each probe takes 0.36 s, 0.91 s, or 2.1 s, respectively, and around 83 % of this is spent for the forward elimination. `Kira` reduces 3044, 10069, and 23670 equations whereas the reconstruction approach selects only 434, 1512, and 4086 mandatory equations.

As last example, we consider `topo5` with a different mass configuration. The non-planar

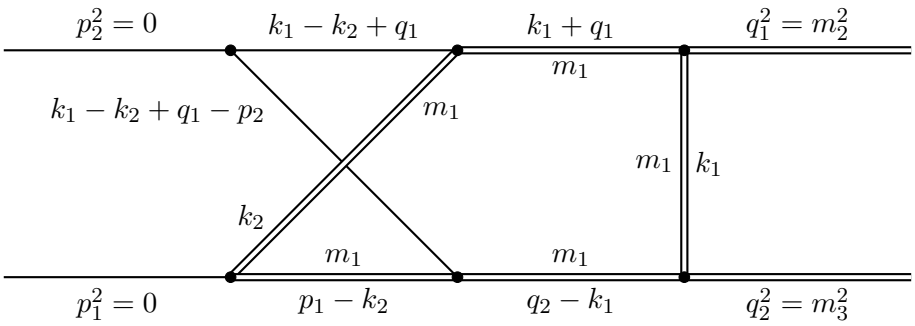

Figure 3: The non-planar double box `topo5_m` which occurs, e.g., in the gluon-induced production of a Higgs and a Z boson.

two-loop diagram shown in Fig. 3 occurs for example in HZ production through gluon fusion. In contrast to `topo5` there are now five massive parameters: m_1, s, m_2, m_3, and t. We can only guess the optimal order from the completed calculations with small $s_{\max}$. There, their highest powers occur in the order stated above, with the additional space-time dimension d at the last position. Thus, we set m_1 to one.

Table 6: Runtime and memory usage for `Kira 1.2` and `Kira` with `FireFly` for `topo5_m` with $r_{\max} = 7$. We use the option `select_mandatory_recursively`, order the variables as m_1, s, m_2, m_3, t, d, and set m_1 to one. The runtime only includes the time spent on the forward elimination and back substitution or the time for the reconstruction, respectively. These tests were run on a computer with two Intel Xeon Gold 6138 and 768 GiB of RAM.

	`Kira 1.2`		`Kira` with `FireFly`		
$s_{\max}$	Runtime	Memory usage	Runtime	CPU time for `pyRed`	Memory usage
1	22 h 40 min	175 GiB	7 h 35 min	88 %	4.7 GiB

Since this example is extremely complicated, we only consider $s_{\max} = 1$ in Tab. 6 to illustrate how both approaches perform. The reconstruction over finite fields is faster by a factor of four and still mainly limited by `pyRed`. Additionally, it requires one to two orders less memory than the algebraic approach. It selects only the 1785 mandatory equations in contrast to the 3140 equations `Kira` reduces. About $1.5 \cdot 10^6$ black-box probes and two prime fields are needed to reconstruct all coefficients. Each `pyRed` solution takes 0.32 s, where the forward elimination contributes with 93 %.

4.3 Application to the perturbative gradient flow

The gradient-flow formalism [70–72] has proven to be a useful tool in lattice QCD. Additionally, there are several areas of potential cross fertilization between lattice and perturbative calculations. However, the perturbative side was only computed to a low order in perturbation theory in many cases. We want to use the IBP reduction as a systematic approach to higher-order calculations. As a verification of this approach we want to compute the same

observable as in Ref. [73], where the computation was performed by solving all integrals numerically. Furthermore, we want to compute two additional observables. Our results will be published in Ref. [74].

Our setup produces all Feynman diagrams, inserts the Feynman rules, and reduces the tensor structures. We also expand the integrals to the leading order in the quark masses and end up with massless three-loop vacuum integrals of the form

$$I(d,t,\{t_f^{\text{up}}\},\{T_i\},\{a_i\}) = \left(\prod_{f=1}^{F}\int_0^{t_f^{\text{up}}} \mathrm{d}t_f\right)\int_{k_1,k_2,k_3} \frac{\exp[-(T_1q_1^2+\cdots+T_6q_6^2)]}{q_1^{2a_1}\cdots q_6^{2a_6}}\,, \tag{43}$$

where $F \leq 4$ is the number of flow-time integrations and the upper limits t_f^{up} are either the flow-time scale t of the diagram or other flow-time variables $t_{f'}$. The T_i are nonnegative linear combinations of the flow variables, i.e. $T_1 = t + t_1 - 2t_3$. Due to the expansion in the quark masses, the flow-time scale t is the only massive scale appearing in the integrals.

Instead of evaluating the integrals numerically as in Ref. [73] we build a system of equations using IBP-like relations, where we set $t = 1$ and, thus, the space-time dimension d is the only parameter. However, the algebraic forward elimination with `Kira 1.2` fills all our available memory of 768 GiB of RAM after running for five days and nine hours on the 80 threads of two Intel Xeon Gold 6138. The progress so far indicates that it probably would have to run for at least several days more if we had enough memory. It might be possible to complete the reduction by adapting the sectorwise forward elimination introduced in `Kira 1.2` for the gradient flow integrals.

Our implementation with finite-field-interpolation techniques as described in Sect. 4.1 runs about two days and 20 hours on ten threads to obtain the complete analytic result. Each numerical system requires around 70 GiB of RAM and takes $\sim 50\,\text{s}$ to parse all analytic coefficients to numerical values, $\sim 10400\,\text{s}$ to complete the forward elimination, and $\sim 16\,\text{s}$ to complete the back substitution. We need three 63-bit prime numbers and 201 numerical solutions in total to reconstruct all rational functions with all rational coefficients and an additional solution with a fourth prime to verify the results.

The highest degree in the numerator is 32 and the highest degree in the denominator 34. At intermediate stages of the analytic forward elimination monomials with at least degree 508 occur.[4] Therefore, this is either a prime example for circumventing large intermediate expressions or of the possibility to neglect large coefficients which are not required for the desired result. However, this is a special problem with only one massive scale, which can be set to one, and, thus, only the space-time dimension d appears in the rational functions. The reduction is also heavily limited by the forward elimination. While this is the default case for numerical solutions, this also holds for the analytic reduction of gradient-flow integrals based on our experience. This is unusual for the reduction of Feynman integrals.

The reduction allows us to express the 3195 required integrals through 188 master integrals. We checked that our reduction yields the correct results by comparing the numerical values for the integrals expressed through the master integrals with the numerical values of the integrals computed directly.

[4] We know this because we had to change the `Fermat` [69] settings to support higher degrees with `Kira 1.1`.

5 Conclusions

We presented the open-source `C++` library `FireFly` for the reconstruction of polynomials and rational functions, which is based on algorithms developed in computer science. Additionally, we made some modifications to reduce the number of black-box probes and avoid cancellations. We showed that `FireFly` is capable to reconstruct multivariate rational functions with many variables.

As an example, we applied `FireFly` to the Laporta algorithm for IBP reductions by linking it to `Kira`. In the example topologies, this approach proved to be competitive to the algebraic approach with `Kira`. `Kira` is usually faster for smaller problems. However, when the system of equations becomes huge and large intermediate expressions occur, the reconstruction with `FireFly` can become faster and require much less memory. The reconstruction approach does not need to compute the large intermediate expressions and can also strictly select only those coefficients which are requested, whereas the algebraic approach can neither circumvent the large intermediate expressions nor perform such a strict selection.

In both approaches, the reduction can be distributed to several computers or several sessions on the same machine, which require less memory. `Kira` allows to perform the back substitution for subsets of master integrals, which is of course limited by the number of master integrals. The reconstruction approach allows to reconstruct subsets of the coefficients or in principle each coefficient on its own. The number of coefficients is usually orders of magnitude larger than the number of master integrals. This also offers the possibility to choose the subsets such that all of them require a sparser shift than the complete set, which can speed up the reconstruction for the individual coefficients significantly. Even for the straight-on approach for quite simple problems, the reconstruction benefits much more from additional CPU threads at least up to several hundred of them. This scaling can even be improved by guessing the required black-box probes in advance. However, this could lead to unnecessary calculations.

Our implementation within `Kira` is strongly limited by the numerical solutions with `pyRed`, which require most of the CPU time in our examples. Therefore, it is desirable to reduce the number of black-box probes required for the reconstruction by algorithmic improvements in `FireFly`. We hope that racing the Ben-Or/Tiwari algorithm against Newton's algorithm as part of Zippel's algorithm as proposed in Refs. [7, 8] achieves this. Optimizations for `pyRed` would also help to widen this bottleneck. We also noted that performing the forward elimination is usually the most expensive step for `pyRed`. Since the algebraic forward elimination is often cheap compared to the back substitution, it might be worthwhile to perform it algebraically and perform only the back substitution numerically for the reconstruction if one expects to require many black-box probes.

Of course, `FireFly` can also be used for hybrid approaches. For example, `Kira 1.2` already offers a multivariate Newton interpolation for (sub-)coefficients over $\mathbb{Z}$ instead of multiplying them algebraically if this seems heuristically useful [40].

Acknowledgments

We are grateful to Robert V. Harlander, Philipp Maierhöfer, Mario Prausa, Benjamin Summ, Johann Usovitsch, and Alexander Voigt for technical support, useful discussions, and comments on the manuscript. This work was supported by the *Deutsche Forschungsgemeinschaft (DFG)* Collaborative Research Center TRR 257 "Particle Physics Phenomenology after the Higgs Discovery" funded through project 396021762. F.L. acknowledges financial support by DFG through project 386986591.

The Feynman diagrams in this paper were drawn with Ti*k*Z-Feynman [75].

A Algorithms

Maximal Quotient Rational Reconstruction

Wang's algorithm for the RR, Alg. (1) [18], only succeeds if the modulus of both numerator $|b|$ and denominator $|c|$ of the rational number are smaller than $\sqrt{p/2}$. However, the proof of the uniqueness of the result only requires $p \geq 2|b||c|$ [14]. This bound is equally distributed to b and c in Alg. (1). In principle, one can adjust the bounds for every number separately, but this requires knowledge of the number before the RR.

Monagan observed that the guess of the rational number comes together with a huge quotient in the Euclidean algorithm [58]. Based on this observation he suggested the algorithm called Maximal Quotient Rational Reconstruction, which returns the rational number corresponding to the largest quotient encountered.

Algorithm 4 Maximal Quotient Rational Reconstruction (MQRR) [58].

Input: A finite field member `a`, the defining prime of the finite field `p`, and the error tolerance parameter `T`.
Output: If succeeded a unique value of `a` in $\mathbb{Q}$ else an error.

```
function MQRR(a, p)
    if a == 0 then
        if p > T then
            return 0;
        else
            Throw an error that the rational reconstruction failed;
        end if
    end if
    n ← 0;   d ← 0;
    t ← 1;   old_t ← 0;
    r ← a;   old_r ← p;
    while r ≠ 0  and  old_r > T do
        quotient ← ⌊ old_r / r ⌋;
        if quotient > T then
            n ← r;   d ← t;   T ← quotient;
        end if
        (old_r , r) ← (r, old_r - quotient * r);
        (old_t, t) ← (t, old_t - quotient * t);
    end while
    if d == 0  or  gcd(n, d) ≠ 1 then
        Throw an error that the rational reconstruction failed;
    end if
    return n / d;
end function
```

It can only be proven that it returns a unique solution if $|b||c| \leq \sqrt{p}/3$. However, it performs much better in the average case because large quotients from random input are rare. The

parameter T allows to choose a lower bound for the quotients and the algorithm fails if the quotients are smaller. Of course, it has to increase with an increasing prime number p, because a random input will generate higher quotients as well. Monagan suggested to calculate it by the formula $T = 2^c\lceil \log_2 p \rceil$ and choose c according to the required error tolerance. We choose $c = 10$ for `FireFly`, which roughly corresponds to 1 % false positive reconstructions.

Solving modified transposed Vandermonde systems

The solution of modified transposed Vandermonde systems defined in Eq. (15) only requires small changes compared to the solution strategies for usual transposed Vandermonde systems given by Refs. [3,5]. Vandermonde matrices of size $M \times M$ are completely determined by M evaluation points $v_1, \ldots, v_M$. Their M^2 components are given by integer powers v_i^j, $i, j = 1, \ldots, M$ in the modified case. For simplicity, we only consider one variable. The solution method of such systems is closely related to Lagrange's polynomial interpolation formula.

Let $B_j(z)$ be the polynomial of degree M defined by

$$B_j(z) = \frac{z}{v_j} \prod_{\substack{m=1 \\ m \neq j}}^{M} \frac{z - v_m}{v_j - v_m} = \sum_{k=1}^{M} A_{jk} z^k. \tag{44}$$

A_{ij} is the matrix defined by the coefficients that arise when the product of Eq. (44) is multiplied out and like terms are collected. The polynomial $B_j(z)$ is specifically designed so that it vanishes at all v_i with $i \neq j$ and has a value of unity at $z = v_j$. Inserting v_i as an argument, one observes

$$B_j(v_i) = \delta_{ij} = \sum_{k=1}^{M} A_{jk} v_i^k. \tag{45}$$

Eq. (45) states that A_{jk} is exactly the inverse of the matrix of components v_i^k. Therefore, the solution of the modified Vandermonde system in Eq. (15) is just that inverse times the right-hand side,

$$c_{\alpha_i} = \sum_{k=1}^{M} A_{ik} f(\vec{y}^{\,k}). \tag{46}$$

It is left to multiply the monomial terms in Eq. (44) out in order to get the components of A_{jk}. By defining a master polynomial $B(z)$ by

$$B(z) \equiv \prod_{m=1}^{M} (z - v_m) = \sum_{i=0}^{M} d_i z^i \tag{47}$$

one can evaluate its coefficients and then obtain the specific B_j via synthetic divisions by the one supernumerary term. Each division is of $O(M)$ and the total procedure is thus of $O(M^2)$. To generalize the solution algorithm to a multivariate system one just needs to redefine v_i as defined in Eq. (15). The algorithm for generalized systems is summarized in Alg. (5).

Algorithm 5 Algorithm to solve modified Vandermonde systems motiviated by Refs. [3,5].

Input: An array of probes of a polynomial, an array of the corresponding inserted values and an array of the contributing degrees of the polynomial ordered (co)lexicographically.

Output: An array with the coefficients of the contributing degrees.

```
function SOLVE_MOD_TRANSPOSED_VANDERMONDE(probes, values, degrees)
    num_eqn ← probes length;  cis;  vis;
    i ← 0;
    for i < length of values do // Calculate v_i
        j ← 0;
        vi ← 1;
        for j < length of values[j] do
            vi ← vi * values[i][j] ** degrees[i][j];
            j ← j + 1;
        end for
        vis[i] = vi;
        i ← i + 1;
    end for
    if num_eqn is equal to 1 then
        cis[0] ← probes[0] / vi;
    else
        dis;
        dis[num_eqn - 1] = -vis[0]; // Calculate d_i
        i ← 1;
        for i < num_eqn do
            j ← num_eqn - 1 - i;
            for j < num_eqn - 1 do
                cis[j] ← dis[j] - vis[i] * dis[j + 1];
                j ← j + 1;
            end for
            dis[num_eqn - 1] ← dis[num_eqn - 1] - vis[0];
            i ← i + 1;
        end for
    end if
    i ← 0;
    for i < num_eqn do // Calculate A_kj and c_α_i
        t ← 1;   b ← 1;   s ← probes[num_eqn - 1];   j ← num_eqn - 1;
        for j > 0 do
            b ← dis[j] + vis[i] * b;
            s ← s + probes[j - 1] * b;
            t ← vis[i] * t + b;
        end for
        cis[i] = s / t / vis[i];
    end for
    return cis;
end function
```

B Internals of `FireFly`

`FireFly` provides additional classes to the ones introduced in Sect. 3, which can be useful for the usage of `FireFly` or modifications of the parallelization. We briefly summarize them in the following:

`BaseReconst`

The base class of functional-reconstruction objects `BaseReconst` serves as a container class for different member variables and functions. Its members are summarized in the following:

`FFInt get_rand()`
: Returns a random number in the current prime field using an implementation of the `pcg_32` algorithm [76].

`uint32_t get_num_eqn()`
: Returns the number of equations that have to be solved at the current stage as a `uint32_t`, which corresponds to the number of black-box probes for the current z_i order. This option is currently only used for `RatReconst`.

`bool is_done()`
: Returns `true` if the interpolation is done.

`bool is_new_prime()`
: Returns `true` if the interpolation object needs a new prime field.

`uint32_t get_prime()`
: Returns the prime counter of the interpolation object.

`vector<uint32_t> get_zi_order()`
: Returns a vector which contains the current tuple of z_i orders.

`uint32_t get_zi()`
: Returns the i of the corresponding z_i which is currently interpolated.

`PolyReconst`

The polynomial-reconstruction class `PolyReconst` is derived from the `BaseReconst` class and performs the functional interpolation and promotion of the polynomial coefficients to $\mathbb{Q}$. Its member variables and functions can be summarized as:

`PolyReconst(uint32_t n, const int deg_inp)`
: Creates a `PolyReconst` object for `n` variables. The second argument `deg_inp` is optional and sets an upper bound on the maximal degree of the black-box function to optimize its interpolation.

`void feed(const FFInt& num, const vector<uint32_t>& fed_zi_ord, \`
`uint32_t fed_prime)`
Feeds the object with a numerical probe `num` of the black-box function, the corresponding z_i order `fed_zi_ord` and prime counter `fed_prime`.

`void interpolate()`
Starts the interpolation for all stored feeds.

`Polynomial get_result()`
Returns the reconstructed polynomial in $\mathbb{Q}$ as a `Polynomial` object. It should only be called after `is_done()` returns `true`.

`void generate_anchor_points()`
Generates the initial values for all z_i.

`FFInt get_rand_zi(uint32_t zi, uint32_t order)`
If already stored, returns the randomized value of z_i corresponding to its z_i order `order`.

`vector<FFInt> get_rand_zi_vec(const vector<uint32_t>& orders)`
If already stored, returns the randomized values of z_i corresponding to their z_i orders `orders` as a vector.

`static void reset()`
Resets all static varibles to start new reconstruction jobs over different prime fields.

RatReconst

The rational-function reconstruction class `RatReconst` is derived from the `BaseReconst` class and performs the functional interpolation and promotion of the rational-function coefficients to $\mathbb{Q}$. Its member variables and functions can be summarized as:

`RatReconst(uint32_t n)`
Creates a `RatReconst` object for `n` variables.

`void feed(const FFInt& new_ti, const FFInt& num, \`
`const vector<uint32_t>& fed_zi_ord, const uint32_t fed_prime)`
Feeds the object with a numerical value of the homogenization variable `new_ti`, a numerical probe `num` of the black-box function, the corresponding z_i order `fed_zi_ord` and prime counter `fed_prime`.

`bool interpolate()`
Starts the interpolation for all stored feeds. Returns `true` if the interpolation is already running.

`RationalFunction get_result()`
Returns the reconstructed rational function in $\mathbb{Q}$ as a `RationalFunction` object. If called when `is_done` returns `false` it exits with an error message.

`void generate_anchor_points()`
Generates the initial values for all z_i with $i > 1$.

`void disable_shift()`
Sets all entries of the static variable `shift` to zero.

`FFInt get_rand_zi(uint32_t zi, uint32_t order)`
If already stored, returns the randomized value of z_i corresponding to its z_i order `order`.

`vector<FFInt> get_rand_zi_vec(const vector<uint32_t>& orders)`
If already stored, returns the randomized values of z_i corresponding to their z_i orders `orders` as a vector.

`vector<FFInt> get_zi_shift(uint32_t zi)`
Returns the shift for a specific z_i.

`vector<FFInt> get_zi_shift_vec()`
Returns the shift for all z_i stored in a vector starting with z_1 at position 0.

`bool need_shift()`
Returns `true` if the objects needs the shift to interpolate the black box over additional prime fields.

`void set_tag(const string& tag)`
Allows the user to set a tag for a `RatReconst` object. If a tag is set, the state of the object will be written to `./ff_save/tag_<prime_counter>.txt`, where `prime_counter` is the currently used entry of the provided prime array. The written file can then be used to resume an interpolation starting from this state.

`void start_from_saved_file(string file_path)`
Passing the absolute path `file_path` to a previously saved state of a `RatReconst` object, `FireFly` will continue the interpolation from this state.

`void scan_for_sparsest_shift()`
Enables the scan variable for a sparser shift. Afterwards a normal interpolation takes place until one accepts a tested shift.

`bool is_shift_working()`
Returns `true` if the current shift is sufficient to obtain a unique normalization. Additionally, all variables are resetted to probe another shift. Should be called during a scan after each interpolation.

`set_zi_shift(const vector<uint32_t>& shifted_zis)`
Set a new, random shift according to the vector `shifted_zis`. `shifted_zis` is filled with zeros and ones to indicate which variable should be shifted, i.e. {`0,1,0`} leads to a shift in z_2. This function has to be called after `is_shift_working()` and before the first interpolation after shifting all variables takes place.

`void accept_shift()`
Accepts the previously tested shift for the reconstruction of the black box.

`static void reset()`
Resets all static varibles to start new reconstruction jobs over different prime fields.

`RationalNumber`

A `RationalNumber` object is a container class which represents an element of $\mathbb{Q}$. The element is built by two arbitrarily precise integer values which are realized through `mpz_class` objects. The latter can be accessed by the member variables `numerator` and `denominator`. For convenience, it also provides the `<<` insertion operator for streams. Useful member functions are summarized in the following:

`RationalNumber(const mpz_class& numerator, const mpz_class& denominator)`
Creates a `RationalNumber` object with a given `numerator` and `denominator`.

`Polynomial`

A `Polynomial` object represents a multivariate polynomial over $\mathbb{Q}$. The data structure is an `unordered_map` called `coefs`, whose key is a `vector<uint32_t>` of length n where n is the number of variables. The entries of this vector are the degrees of each variable. This vector is the equivalent of the multi index α_i introduced in Eq. (3). The values of the map are `RationalNumber` objects and correspond to the coefficient c_{α_i} as defined in Eq. (3). For convenience, it provides the `<<` insertion operator for streams. Useful member functions are summarized in the following:

`Polynomial(const unordered_map<vector<uint32_t>, RationalNumber>& coefs)`
Creates a `Polynomial` object with the given degrees and coefficients of `coefs`.

`void clear()`
Clears the polynomial.

`string to_string(const vector<string>& symbols)`
Rewrites the polynomial by incorporating the given `symbols` and returns it as a `string`.

`void sort()`
Sorts the coefficient map of the polynomial in a lexicographical order with respect to the degrees.

`RationalFunction`

A `RationalFunction` object is a container class which combines two `Polynomial` objects to a rational function by defining a `numerator` and `denominator`. For convenience, it provides the `<<` insertion operator for streams. Useful member functions are:

`RationalFunction(const Polynomial& numerator, const Polynomial&\`
`  denominator)`
Creates a `RationalFunction` object for the given `Polynomial` objects.

`string to_string(const vector<string>& symbols)`
Rewrites the rational function incorporating the given `symbols` and returns it as a `string`.

`ReconstHelper`

The `ReconstHelper` file contains the hundred largest 63-bit prime numbers which can be accessed through the function `primes()` that returns a vector. The first element is the largest one and the last is the smallest.

`utils`

The `utils` file contains the implementation of some algorithms and a function to generate all possible tuples of zeros and ones needed for a full scan of all possible shifts with respect to the number of variables.

```
pair<mpz_class, mpz_class> run_chinese_remainder(const pair<mpz_class, \
    mpz_class>& p1, const pair<mpz_class, mpz_class>& p2)
```
Combines the two pairs `p1` and `p2` of a number and a prime number to a new pair with the CRT given by Alg. (2).

```
pair<bool, RationalNumber> get_rational_coef(const mpz_class& a, \
    const mpz_class& p)
```
Applies Alg. (1) for the integer `a` and the prime number `p`. It returns the `pair` of a `bool` which indicates if it suceeded and the reconstructed `RationalNumber`.

```
pair<bool, RationalNumber> get_rational_coef_mqrr(const mpz_class& a, \
    const mpz_class& p)
```
Applies Alg. (4) for the integer `a` and the prime number `p`. It returns the `pair` of a `bool` which indicates if it suceeded and the reconstructed `RationalNumber`.

```
vector<vector<uint32_t>> generate_possible_shifts(uint32_t r))
```
Returns a vector of all possible tuples of zeros and ones of length `r` ordered from tuples containing only one one to at most `r - 1` ones.

Parallelization

Let us finish this section by a few more words on the details of parallel computing. Each `PolyReconst` and `RatReconst` object possesses two mutexes, namely, `mutex_status` and `mutex_statics`. The former protects the status of the object, i.e. all member variables which can be accessed through getters. All changes to these variables only occur after `mutex_status` was obtained and all getters obtain `mutex_status` before accessing the variables. `mutex_statics` protects the static variables which are used by all reconstruction objects, e.g. `shift` and `rand_zi`, which are private members representing the used shift and a collection of numerical values for different z_i orders.

To interpolate a function, we follow a two step paradigm. In a first step, we provide the reconstruction class with a black-box probe at a given z_i order. This procedure is called a *feed*. If a reconstruction is not already done, the reconstruction class always accepts a feed if it is evaluated for the current prime number. All feeds are stored in a queue for later usage. The interpolation and reconstruction is done independently of the feed procedure. If the member function `interpolate` is called, the class tries to reconstruct the black-box function with all stored feeds in the queue and runs until it is done or a probe corresponding to a z_i order is requested, which is not present in the queue. During the interpolation, the object can be fed, but no additional interpolation job is accepted. Thus, the `interpolate` function checks whether another interpolation is already running and returns immediately should this be the case. Otherwise it takes feeds from the queue and proceeds with the interpolation until the queue is empty or the reconstruction is done. This procedure allows the user to start the interpolation for every feed without concerning whether the interpolation is already running or not.

References

[1] M. Abramowitz and I.A. Stegun, *Handbook of Mathematical Functions With Formulas, Graphs, and Mathematical Tables*, First Edition, Dover Publications, 1964.

[2] R. Zippel, *Probabilistic algorithms for sparse polynomials*, *Symbolic Algebraic Comp.. EUROSAM* **1979**, 216 (1979).

[3] R. Zippel, *Interpolating Polynomials from their Values*, *J. Symb. Comp.* **9**, 375 (1990).

[4] M. Ben-Or and P. Tiwari, *A Deterministic Algorithm for Sparse Multivariate Polynomial Interpolation*, *Proc. ACM Symp. Theory Comp.* **20**, 301 (1988).

[5] E. Kaltofen and Lakshman Y., *Improved Sparse Multivariate Polynomial Interpolation Algorithms*, *Symbolic Algebraic Comp.. ISSAC* **1988**, 467 (1989).

[6] E. Kaltofen, Lakshman Y.N., and J.-M. Wiley, *Modular Rational Sparse Multivariate Polynomial Interpolation*, *Proc. Int. Symp. Symbolic Algebraic Comp.* **1990**, 135 (1990).

[7] E. Kaltofen, W.-s. Lee, and 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**, 192 (2000).

[8] E. Kaltofen and W.-s. Lee, *Early termination in sparse interpolation algorithms*, *J. Symb. Comp.* **36**, 365 (2003).

[9] S.M.M. Javadi and M. Monagan, *Parallel Sparse Polynomial Interpolation over Finite Fields*, *Proc. Int. Workshop Parallel Symbolic Comp.* **4**, 160 (2010).

[10] D.Y. Grigoriev, M. Karpinski, and M.F. Singer, *Interpolation of Sparse Rational Functions Without Knowing Bounds on Exponents*, *Proc. Symp. Foundations Comp. Sci.* **31**, 840 (1990).

[11] D.Y. Grigoriev and M. Karpinski, *Algorithms for Sparse Rational Interpolation*, *Proc. Int. Symp. Symbolic Algebraic Comp.* **1991**, 7 (1991).

[12] D. Grigoriev, M. Karpinski, and M.F. Singer, *Computational Complexity of Sparse Rational Interpolation*, *SIAM J. Comp.* **23**, 1 (1994).

[13] S. Khodadad and M. Monagan, *Fast Rational Function Reconstruction*, *Proc. Int. Symp. Symbolic Algebraic Comp.* **2006**, 184 (2006).

[14] J. von zur Gathen and J. Gerhard, *Modern Computer Algebra*, Third Edition, Cambridge University Press, 2013.

[15] E. Kaltofen and Z. Yang, *On Exact and Approximate Interpolation of Sparse Rational Functions*, *Proc. Int. Symp. Symbolic Algebraic Comp.* **2007**, 203 (2007).

[16] A. Cuyt and W.-s. Lee, *Sparse interpolation of multivariate rational functions*, *Theor. Comp. Sci.* **412**, 1445 (2011).

[17] Q.-L. Huang and X.-S. Gao, *Sparse Rational Function Interpolation with Finitely Many Values for the Coefficients*, *Math. Aspects Comp. Information Sci.* **2017**, 227 (2017), arXiv:1706.00914 [cs.SC].

[18] P.S. Wang, *A p-adic Algorithm for Univariate Partial Fractions*, *Proc. ACM Symp. Symbolic Algebraic Comp.* **1981**, 212 (1981).

[19] F.V. Tkachov, *A theorem on analytical calculability of 4-loop renormalization group functions*, *Phys. Lett.* **B 100**, 65 (1981).

[20] K.G. Chetyrkin and F.V. Tkachov, *Integration By Parts: The algorithm to calculate beta functions in four loops*, *Nucl. Phys.* **B 192**, 159 (1981).

[21] S.G. Gorishny, S.A. Larin, L.R. Surguladze, and F.V. Tkachov, *Mincer: Program for multiloop calculations in quantum field theory for the Schoonschip system*, *Comp. Phys. Commun.* **55**, 381 (1989).

[22] S.A. Larin, F.V. Tkachov, and J.A.M. Vermaseren, *The FORM version of MINCER*, 1991.

[23] D.J. Broadhurst, *Three loop on-shell charge renormalization without integration:* $\Lambda^{\overline{MS}}_{QED}$ *to four loops*, *Z. Phys.* **C 54**, 599 (1992).

[24] M. Steinhauser, *MATAD: a program package for the computation of massive tadpoles*, *Comp. Phys. Commun.* **134**, 335 (2001), hep-ph/0009029.

[25] K. Melnikov and T. van Ritbergen, *Three-Loop Slope of the Dirac Form Factor and the* $1S$ *Lamb Shift in Hydrogen*, *Phys. Rev. Lett.* **84**, 1673 (2000), hep-ph/9911277.

[26] K. Melnikov and T. van Ritbergen, *The three-loop relation between the* $\overline{MS}$ *and the pole quark masses*, *Phys. Lett.* **B 482**, 99 (2000), hep-ph/9912391.

[27] K. Melnikov and T. van Ritbergen, *The three-loop on-shell renormalization of QCD and QED*, *Nucl. Phys.* **B 591**, 515 (2000), hep-ph/0005131.

[28] R.N. Lee, *Presenting LiteRed: a tool for the Loop InTEgrals REDuction*, arXiv:1212.2685 [hep-ph].

[29] A.V. Smirnov and V.A. Smirnov, *FIRE4, LiteRed and accompanying tools to solve integration by parts relations*, *Comp. Phys. Commun.* **184**, 2820 (2013), arXiv:1302.5885 [hep-ph].

[30] R.N. Lee, *LiteRed 1.4: a powerful tool for reduction of multiloop integrals*, *J. Phys. Conf. Ser.* **523**, 012059 (2014), arXiv:1310.1145 [hep-ph].

[31] B. Ruijl, T. Ueda, and J.A.M. Vermaseren, FORCER, *a* FORM *program for the parametric reduction of four-loop massless propagator diagrams*, arXiv:1704.06650 [hep-ph].

[32] S. Laporta, *High-precision calculation of multi-loop Feynman integrals by difference equations*, *Int. J. Mod. Phys.* **A 15**, 5087 (2000), hep-ph/0102033.

[33] C. Anastasiou and A. Lazopoulos, *Automatic Integral Reduction for Higher Order Perturbative Calculations*, *JHEP* **0407**, 046 (2004), hep-ph/0404258.

[34] A.V. Smirnov, *Algorithm FIRE – Feynman Integral REduction*, *JHEP* **0810**, 107 (2008), arXiv:0807.3243 [hep-ph].

[35] A.V. Smirnov, *FIRE5: a C++ implementation of Feynman Integral REduction*, *Comp. Phys. Commun.* **189**, 182 (2015), arXiv:1408.2372 [hep-ph].

[36] A.V. Smirnov and F.S. Chukharev, *FIRE6: Feynman Integral REduction with Modular Arithmetic*, arXiv:1901.07808 [hep-ph].

[37] C. Studerus, *Reduze - Feynman Integral Reduction in C++*, *Comp. Phys. Commun.* **181**, 1293 (2010), arXiv:0912.2546 [physics.comp-ph].

[38] A. von Manteuffel and C. Studerus, *Reduze 2 - Distributed Feynman Integral Reduction*, arXiv:1201.4330 [hep-ph].

[39] P. Maierhöfer, J. Usovitsch, and P. Uwer, *Kira - A Feynman Integral Reduction Program*, *Comp. Phys. Commun.* **230**, 99 (2018), arXiv:1705.05610 [hep-ph].

[40] P. Maierhöfer and J. Usovitsch, *Kira 1.2 Release Notes*, arXiv:1812.01491 [hep-ph].

[41] M. Kauers, *Fast solvers for dense linear systems*, *Nucl. Phys. Proc. Suppl.* **183**, 245 (2008).

[42] P. Kant, *Finding Linear Dependencies in Integration-By-Parts Equations: A Monte Carlo Approach*, *Comp. Phys. Commun.* **185**, 1473 (2014), arXiv:1309.7287 [hep-ph].

[43] A. von Manteuffel and R.M. Schabinger, *A novel approach to integration by parts reduction*, *Phys. Lett.* **B 744**, 101 (2015), arXiv:1406.4513 [hep-ph].

[44] T. Peraro, *Scattering amplitudes over finite fields and multivariate functional reconstruction*, *JHEP* **1612**, 030 (2016), arXiv:1608.01902 [hep-ph].

[45] A. von Manteuffel and R.M. Schabinger, *Quark and gluon form factors to four-loop order in QCD: the N_f^3 contributions*, *Phys. Rev.* **D 95**, 034030 (2017), arXiv:1611.00795 [hep-ph].

[46] J.M. Henn, T. Peraro, M. Stahlhofen, and P. Wasser, *Matter dependence of the four-loop cusp anomalous dimension*, arXiv:1901.03693 [hep-ph].

[47] A. von Manteuffel and R.M. Schabinger, *Quark and gluon form factors in four loop QCD: the N_f^2 and $N_{q\gamma}N_f$ contributions*, arXiv:1902.08208 [hep-ph].

[48] A. von Manteuffel and R.M. Schabinger, *Planar master integrals for four-loop form factors*, arXiv:1903.06171 [hep-ph].

[49] S. Badger, C. Brønnum-Hansen, H.B. Hartanto, and T. Peraro, *First Look at Two-Loop Five-Gluon Scattering in QCD*, *Phys. Rev. Lett.* **120**, 092001 (2018), arXiv:1712.02229 [hep-ph].

[50] S. Abreu, F. Febres Cordero, H. Ita, B. Page, and M. Zeng, *Planar Two-Loop Five-Gluon Amplitudes from Numerical Unitarity*, *Phys. Rev.* **D 97**, 116014 (2018), arXiv:1712.03946 [hep-ph].

[51] Z. Bern, J.J. Carrasco, W.-M. Chen, A. Edison, H. Johansson, J. Parra-Martinez, R. Roiban, and M. Zeng, *Ultraviolet Properties of $\mathcal{N} = 8$ Supergravity at Five Loops*, *Phys. Rev.* **D 98**, 086021 (2018), arXiv:1804.09311 [hep-ph].

[52] S. Abreu, F. Febres Cordero, H. Ita, B. Page, and V. Sotnikov, *Planar Two-Loop Five-Parton Amplitudes from Numerical Unitarity*, *JHEP* **1811**, 116 (2018), arXiv:1809.09067 [hep-ph].

[53] S. Badger, C. Brønnum-Hansen, H.B. Hartanto, and T. Peraro, *Analytic helicity amplitudes for two-loop five-gluon scattering: the single-minus case*, *JHEP* **1901**, 186 (2019), arXiv:1811.11699 [hep-ph].

[54] S. Abreu, J. Dormans, F. Febres Cordero, H. Ita, and B. Page, *Analytic Form of Planar Two-Loop Five-Gluon Scattering Amplitudes in QCD*, *Phys. Rev. Lett.* **122**, 082002 (2019), arXiv:1812.04586 [hep-ph].

[55] S. Abreu, L.J. Dixon, E. Herrmann, B. Page, and M. Zeng, *The two-loop five-point amplitude in $\mathcal{N} = 4$ super-Yang-Mills theory*, arXiv:1812.08941 [hep-th].

[56] S. Abreu, L.J. Dixon, E. Herrmann, B. Page, and M. Zeng, *The two-loop five-point amplitude in $\mathcal{N} = 8$ supergravity*, arXiv:1901.08563 [hep-th].

[57] A. Díaz and E. Kaltofen, FOXBOX*: A System for Manipulating Symbolic Objects in Black Box Representation*, *Proc. Int. Symp. Symbolic Algebraic Comp.* **1998**, 30 (1998).

[58] M. Monagan, *Maximal Quotient Rational Reconstruction: An Almost Optimal Algorithm for Rational Reconstruction*, *Proc. Int. Symp. Symbolic Algebraic Comp.* **2004**, 243 (2004).

[59] E. Kaltofen and H. Rolletschek, *Computing Greatest Common Divisors and Factorizations in Quadratic Number Fields*, *Math. Comp.* **53**, 697 (1989).

[60] E. Kaltofen and 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**, 301 (1990).

[61] P.S. Wang, M.J.T. Guy, and J.H. Davenport, *P-adic Reconstruction of Rational Numbers*, *ACM SIGSAM Bulletin* **16, 2**, 2 (1982).

[62] T. Granlund et.al., *The GNU Multiple Precision Arithmetic Library*, https://gmplib.org/.

[63] W. Hart et.al., *FLINT: Fast Library for Number Theory*, http://www.flintlib.org/.

[64] W.B. Hart, *Fast Library for Number Theory: An Introduction*, *Proc. Int. Congress Mathe. Software* **2010**, 88 (2010).

[65] P. Athron, M. Bach, D. Harries, T. Kwasnitza, J.-h. Park, D. Stöckinger, A. Voigt, and J. Ziebell, *FlexibleSUSY 2.0: Extensions to investigate the phenomenology of SUSY and non-SUSY models*, *Comp. Phys. Commun.* **230**, 145 (2018), arXiv:1710.03760 [hep-ph].

[66] A.G. Grozin, *Integration by parts: An introduction*, *Int. J. Mod. Phys.* **A 26**, 2807 (2011), arXiv:1104.3993 [hep-ph].

[67] V.A. Smirnov, *Analytic Tools for Feynman Integrals*, First Edition, Springer-Verlag Berlin Heidelberg, 2012.

[68] A.V. Kotikov and S. Teber, *Multi-loop techniques for massless Feynman diagram calculations*, arXiv:1805.05109 [hep-th].

[69] R.H. Lewis, *Fermat: A Computer Algebra System for Polynomial and Matrix Computation*, http://home.bway.net/lewis/.

[70] M. Lüscher, *Properties and uses of the Wilson flow in lattice QCD*, *JHEP* **1008**, 071 (2010) [Erratum: *JHEP* **1403**, 092 (2014)], arXiv:1006.4518 [hep-lat].

[71] M. Lüscher and P. Weisz, *Perturbative analysis of the gradient flow in non-abelian gauge theories*, *JHEP* **1102**, 051 (2011), arXiv:1101.0963 [hep-th].

[72] M. Lüscher, *Chiral symmetry and the Yang–Mills gradient flow*, *JHEP* **1304**, 123 (2013), arXiv:1302.5246 [hep-lat].

[73] R.V. Harlander and T. Neumann, *The perturbative QCD gradient flow to three loops*, *JHEP* **1606**, 161 (2016), arXiv:1606.03756 [hep-ph].

[74] J. Artz, R.V. Harlander, F. Lange, T. Neumann, and M. Prausa, in preparation.

[75] J. Ellis, *TikZ-Feynman: Feynman diagrams with TikZ*, *Comp. Phys. Commun.* **210**, 103 (2017), arXiv:1601.05437 [hep-ph].

[76] M.E. O'Neill, *PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation*, https://www.cs.hmc.edu/tr/hmc-cs-2014-0905.pdf.