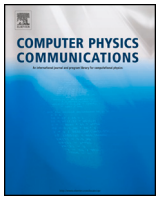
Computer Physics Communications 307 (2025) 109395

Contents lists available at ScienceDirect

Computer Physics Communications

journal homepage: www.elsevier.com/locate/cpc



Computer Programs in Physics

LinApart: Optimizing the univariate partial fraction decomposition ☆

B. Chargeishvili [a], L. Fekésházy [a,b,c,*], G. Somogyi [b], S. Van Thurenhout [b]

[a] II. Institut für Theoretische Physik, Universität Hamburg, Luruper Chaussee 149, 22761, Hamburg, Germany
[b] HUN-REN Wigner Research Centre for Physics, Konkoly-Thege Miklós út 29-33, 1121, Budapest, Hungary
[c] Institute for Theoretical Physics, ELTE Eötvös Loránd University, Pázmány Péter sétány 1/A, 1117, Budapest, Hungary

ARTICLE INFO

Keywords:
Partial fraction decomposition

ABSTRACT

We present LinApart, a routine designed for efficiently performing the univariate partial fraction decomposition of large symbolic expressions. Our method is based on an explicit closed formula for the decomposition of rational functions with fully factorized denominators. We provide an implementation in the WOLFRAM MATHEMATICA language, which can lead to very significant performance gains over the built-in Apart command. Furthermore, a C language library implementing the core functionality and suitable for interfacing with other software is also provided. Both codes are made available at https://github.com/fekeshazy/LinApart.

Program summary

Program title: LinApart

CPC Library link to program files: https://doi.org/10.17632/v8vg2mjjnb.1

Developer's repository link: https://github.com/fekeshazy/LinApart

Licensing provisions: MIT license

Programming language: WOLFRAM MATHEMATICA and C

Nature of problem: Analytic computations in quantum field theory often produce large expressions involving complicated rational functions. In many cases, such as symbolic integration, performing partial fraction decomposition of the latter is vital. Even though univariate partial fraction decomposition is conceptually well-understood, the complexity of state-of-the-art calculations is such that use of readily available tools, such as the Apart command in MATHEMATICA, becomes highly impractical due to time and memory constraints. Here we present LinApart, a routine developed for performing univariate partial fraction decomposition in a highly efficient manner. Improvements in timing and memory usage of up to five orders of magnitude are achieved for rational expressions with as few as ten denominator factors. As such, the routine allows to perform decomposition problems that were previously intractable.

Solution method: The LinApart routine implements a simple closed form expression for univariate partial fraction decomposition of rational functions with fully factored denominators based on the residue theorem. The MATHEMATICA implementation makes use of the highly efficient built-in differentiation routine to evaluate this formula. The C language implementation employs a version of the formula that does not require symbolic differentiation but is rather based on the computation of multinomial coefficients.

Additional comments including restrictions and unusual features: The routine is based on a formula which assumes that the denominator of the rational function to be decomposed is in a fully factorized form, i.e., it is a product whose factors are linear in the decomposition variable. For maximum efficiency, the MATHEMATICA implementation by default does not perform any factorization and non-linear denominators, if present, are simply disregarded during the decomposition. Thus, in such cases the output will not be in a fully decomposed form. This issue does not arise in the C implementation, which only accepts linear denominator factors of the form $(x-a)^m$ as input.

☆ The review of this paper was arranged by Prof. Z. Was.

* Corresponding author at: II. Institut für Theoretische Physik, Universität Hamburg, Luruper Chaussee 149, 22761, Hamburg, Germany.
E-mail addresses: bakar.chargeishvili@desy.de (B. Chargeishvili), levente.fekeshazy@desy.de (L. Fekésházy), somogyi.gabor@wigner.hun-ren.hu (G. Somogyi), sam.van.thurenhout@wigner.hun-ren.hu (S. Van Thurenhout).

https://doi.org/10.1016/j.cpc.2024.109395
Received 19 June 2024; Received in revised form 30 August 2024; Accepted 9 October 2024
Available online 15 October 2024

1. Introduction

Partial fraction decomposition is a standard tool commonly employed in many aspects of perturbative quantum field theory (QFT) calculations. Its uses include simplifying complicated expressions as well as bringing them to a unique form for further manipulation. In particular, it is an important step during the analytic computation of loop and phase-space integrals. One possible approach to such calculations relies on deriving multidimensional real Euler-type integral representations for the integrals of interest and sequentially performing the integration over each variable in terms of functions defined as iterated integrals, such as multiple polylogarithms [1–5]. Here partial fraction decomposition of the expression in the integration variable is necessary in order to cast the integral into a form that can be recognized as a multiple polylogarithm. Partial fraction decomposition as well as its multivariate generalizations have also been applied in conjunction with the integration-by-parts method for obtaining simplified forms of intermediate and final results [6–11].

Univariate partial fraction decomposition is a well-understood problem and various algorithms for performing such a decomposition are known, with implementations in many computer algebra systems,[1] see e.g. [13–17]. However these days, the complexity of perturbative QFT problems has reached a point where in order to obtain results, the efficiency of each step of the calculation must be carefully considered. In particular, the direct symbolic computation of phase-space integrals relevant for building subtraction schemes beyond next-to-leading order can lead to expressions where the (lack of) efficiency of standard partial fraction decomposition routines becomes a bottleneck to performing the calculation [18–21].

Hence it is desirable to have a very efficient way of performing the partial fraction decomposition. In this paper we present `LinApart`, a univariate partial fraction decomposition routine and provide an implementation in the WOLFRAM MATHEMATICA language. The routine is based on a simple closed formula following from the residue theorem and leads to massive performance gains over the built-in `Apart` command. This allows one to obtain solutions for a whole range of decomposition problems that were previously intractable. Furthermore, a C language library implementing the core functionality is also made available. This can be interfaced with other computer algebra systems, such as FORM [22,23], that currently lack built-in partial fraction decomposition capabilities. Finally, in order to demonstrate the use of the library, a standalone C program is provided.

The paper is organized as follows. In section 2 we present our basic formula for univariate partial fraction decomposition. In section 3 we discuss the implementation and usage of the `LinApart` routine in WOLFRAM MATHEMATICA and C. Then, in section 4 we highlight the massive performance improvements of the MATHEMATICA implementation with respect to `Apart` on various classes of rational functions. Finally, in section 5 we present our conclusions and outlook.

2. Closed formula for univariate partial fraction decomposition

Consider a rational function of the variable x,[2]

$$R(x) = \frac{P(x)}{Q(x)}, \tag{1}$$

where $P(x)$ and $Q(x)$ are polynomials of degree $\deg P$ and $\deg Q$. $R(x)$ can be represented as a linear combination of monomials of x divided by $Q(x)$,

$$R(x) = \sum_{l=0}^{\deg P} p_l \frac{x^l}{Q(x)}, \tag{2}$$

hence we may restrict our attention to rational functions of the form $f(x) = \frac{x^l}{Q(x)}$. If $f(x)$ is a proper rational function, i.e., $l < \deg Q$, it is well-known that $f(x)$ can be written as

$$f(x) = \sum_{i=1}^{n} \left(\frac{c_{i1}}{x - a_i} + \frac{c_{i2}}{(x-a_i)^2} + \cdots + \frac{c_{im_i}}{(x-a_i)^{m_i}} \right), \tag{3}$$

where the index i counts the n distinct roots $\{a_i\}_{i=1}^n$ of the polynomial $Q(x)$ and m_i denotes the multiplicity of the i-th root. We are considering the partial fraction decomposition of $f(x)$ over the complex numbers, such that the polynomial $Q(x)$ can be written as a product of linear factors (in x) with positive powers: $Q(x) = \prod_{i=1}^n (x-a_i)^{m_i}$.[3] Then, the uniqueness of the Laurent series implies that c_{ij} is simply the coefficient of the term $(x-a_i)^{-1}$ in the Laurent expansion of the auxiliary function $g_{ij}(x) = (x-a_i)^{j-1} f(x)$ around the point a_i. In other words, c_{ij} is just the residue of $g_{ij}(x)$ at a_i. This residue can be directly computed as

$$c_{ij} = \mathrm{Res}(g_{ij}, a_i) = \frac{1}{(m_i - j)!} \lim_{x \to a_i} \frac{d^{m_i - j}}{dx^{m_i - j}} \Big((x-a_i)^{m_i} f(x) \Big). \tag{4}$$

Since

$$(x-a_i)^{m_i} f(x) = x^l \prod_{\substack{k=1 \\ k \neq i}}^{n} \frac{1}{(x-a_k)^{m_k}} \tag{5}$$

is independent of a_i, there is no subtlety in taking the limit and one may simply replace $x \to a_i$ in eq. (4) to obtain an expression for c_{ij} directly in terms of the roots,

[1] Dating back to the 1960's with Veltman's SCHOONSCHIP [12].

[2] Here and in the following we will denote the decomposition variable by x.

[3] We note that the complete factorization of the denominator to linear factors is necessary in applications related to symbolic integration in terms of multiple polylogarithms, since multiple polylogarithms have linear integration kernels.

$$c_{ij} = \frac{1}{(m_i - j)!} \frac{d^{m_i - j}}{da_i^{m_i - j}} a_i^l \prod_{\substack{k=1 \\ k \neq i}}^{n} \frac{1}{(a_i - a_k)^{m_k}} . \tag{6}$$

Hence, $f(x)$ can be expressed as

$$f(x) = \sum_{i=1}^{n} \sum_{j=1}^{m_i} \frac{c_{ij}}{(x - a_i)^j} = \sum_{i=1}^{n} \sum_{j=1}^{m_i} \frac{1}{(x - a_i)^j} \frac{1}{(m_i - j)!} \frac{d^{m_i - j}}{da_i^{m_i - j}} a_i^l \prod_{\substack{k=1 \\ k \neq i}}^{n} \frac{1}{(a_i - a_k)^{m_k}} . \tag{7}$$

This formula can be written in a more compact form by noting that the factor $(x - a_i)^{-j}$ is related to the $(j-1)$-st derivative of $(x - a_i)^{-1}$ with respect to a_i,

$$\frac{1}{(x - a_i)^j} = \frac{1}{(j-1)!} \frac{d^{j-1}}{da_i^{j-1}} \frac{1}{x - a_i} . \tag{8}$$

Substituting eq. (8) into eq. (7), one sees that the summation over j corresponds to the general Leibniz rule for the $(m_i - 1)$-st derivative of a product of two functions,

$$f(x) = \sum_{i=1}^{n} \frac{1}{(m_i - 1)!} \frac{d^{m_i - 1}}{da_i^{m_i - 1}} \left(\frac{a_i^l}{x - a_i} \prod_{\substack{k=1 \\ k \neq i}}^{n} \frac{1}{(a_i - a_k)^{m_k}} \right) . \tag{9}$$

Eq. (9) is in a form which can be used directly in any high-level programming language in which symbolic differentiation is implemented, such as WOLFRAM MATHEMATICA. However, it is also straightforward to perform the differentiation in eq. (9) symbolically, leading to an expression that involves only elementary arithmetic operations and is hence more suitable for implementation in low-level programming languages, such as C. Indeed, using the multinomial generalization of the Leibniz rule,

$$\frac{d^m}{dx^m} \prod_{j=1}^{n} f_j(x) = \sum_{j_1 + \cdots + j_n = m} \binom{m}{j_1 \dots j_n} \prod_{l=1}^{n} \frac{d^{j_l} f_l(x)}{dx^{j_l}} , \tag{10}$$

where $\binom{m}{j_1 \dots j_n} = \frac{m!}{j_1! \dots j_n!}$ is a multinomial coefficient and the sum runs over all values of $j_1, \dots, j_n$ that sum to m, we find

$$f(x) = \sum_{i=1}^{n} \sum_{j_{-1} + j_0 + j_1 + \cdots + \hat{j}_i + \cdots + j_n = m_i - 1} \binom{l}{j_{-1}} \frac{a_i^{l - j_{-1}}}{(x - a_i)^{j_0 + 1}} \prod_{\substack{k=1 \\ k \neq i}}^{n} \binom{m_k + j_k - 1}{j_k} \frac{(-1)^{j_k}}{(a_i - a_k)^{m_k + j_k}} . \tag{11}$$

Here the hat on the index $\hat{j}_i$ denotes that j_i is removed from the set of indices.

Finally, we must also deal with improper rational functions of the form $f(x) = \frac{x^l}{Q(x)}$, where $l \geq \deg Q$. A straightforward way to proceed would be to write $f(x)$ as the sum of a polynomial and a proper rational function using polynomial division, after which the proper rational function part can be decomposed using eq. (9). However, it turns out to be more efficient to perform the polynomial division symbolically in the following way. First, write

$$f(x) = x^{l - (m-1)} \frac{x^{m-1}}{Q(x)} , \qquad l \geq m , \tag{12}$$

where $m = \deg Q(x) = \sum_{i=1}^{n} m_i$ is simply the degree of the denominator. Then, the second factor is a proper rational function by construction and we can apply the formula in eq. (9) to decompose it into partial fractions. After this decomposition, only rational functions of the form $g(x) = \frac{x^p}{(x-a)^q}$ remain. One can then perform the polynomial division symbolically as follows,

$$g(x) = \frac{x^p}{(x - a)^q} = \sum_{i=0}^{p-q} \binom{p}{i} a^i (x - a)^{p-q-i} + \sum_{i=p-q+1}^{p} \binom{p}{i} a^i (x - a)^{p-q-i} . \tag{13}$$

The first term is nonzero only if $p - q \geq 0$ (otherwise this sum is empty) and represents the quotient polynomial, while the second term gives the proper rational function remainder and is in a decomposed form already. Note that it is possible to write the quotient polynomial explicitly, since one can show that

$$\sum_{i=0}^{p-q} \binom{p}{i} a^i (x - a)^{p-q-i} = \sum_{i=0}^{p-q} \binom{p - 1 - i}{q - 1} a^{p-q-i} x^i . \tag{14}$$

In our actual implementation, we prefer to use this second form, i.e.,

$$g(x) = \sum_{i=0}^{p-q} \binom{p - 1 - i}{q - 1} a^{p-q-i} x^i + \sum_{i=p-q+1}^{p} \binom{p}{i} a^i (x - a)^{p-q-i} . \tag{15}$$

Eqs. (11) and (15) thus give explicit formulae representing the partial fraction decomposition of univariate rational functions. As noted above, for the purposes of actual implementation, eq. (9) is also useful in high-level languages where symbolic differentiation is available.

3. Implementation and usage

In this section, we present our implementations of the `LinApart` routine in the WOLFRAM MATHEMATICA and C languages, which can be obtained at https://github.com/fekeshazy/LinApart.

3.1. The WOLFRAM MATHEMATICA routine

The MATHEMATICA implementation is contained in a single file `LinApart.m`. The routine can handle both proper and improper fractions. For proper fractions, the decomposition is directly computed according to eq. (9). Extensive testing has revealed that this symbolic approach, leveraging the efficiency of the built-in differentiation routine, outperforms alternative methods. For improper fractions eq. (12) is first employed to decompose each term into a product of a monomial and a proper rational function, followed by the application of eq. (15). By default, the resulting decomposition is returned without any further simplification or gathering of terms. However, if present, coefficients which are independent of the decomposition variable are distributed over the decomposed form using the built-in `Distribute` command.

The derivations of the basic formulae assume that the denominator is in a fully factorized form, i.e., each denominator factor is linear in x, and that each multiplicity m_i is a positive integer. Nevertheless, the implementation accepts any valid MATHEMATICA expression as input, including sums of several functions, non-linear factors in the denominator, non-integer exponents and functions of x that are not rational. Sums in the input are treated by applying partial fraction decomposition term-by-term. Next, each term is split into a true rational function part with linear denominators and a left-over piece, which contains all non-linear denominators, factors with non-integer exponents and non-rational functions of x. For the purposes of this splitting, linear denominators with rational number exponents $r \in \mathbb{Q}$ are treated as follows. First we write

$$(x-a)^r = (x-a)^{\lfloor r \rfloor} \cdot (x-a)^{r-\lfloor r \rfloor}\,, \tag{16}$$

where $\lfloor r \rfloor$ denotes the floor function, i.e., the greatest integer less than or equal to r. The exponent of the first factor is an integer by construction, and so this factor is included in the rational function part. The exponent of the second factor is a rational number, strictly smaller than one and this factor enters the left-over piece. This prescription is adopted in order to reproduce the behaviour of the `Apart` command on such expressions. The rational function part is then decomposed as explained above, while the irrational part is left unchanged.

In some practical applications, the partial fraction decomposition must be performed on large expressions with many terms. In such cases, significant performance gains can be obtained by processing the input prior to the decomposition. Examples of such operations include

- Factorization: prior to decomposition, individual terms of the input can be factored over either the integers or Gaussian integers, depending on the desired domain. Thus, non-linear denominators that factorize over these domains will also be considered during the decomposition process.
- Collection of terms before decomposition: terms with identical x-dependence can be grouped together, reducing the number of individual decompositions required.

These pre-processing operations are made available in our implementation as options, to be introduced below. By incorporating these features, the routine provides a robust and versatile framework for partial fraction decomposition, allowing the user to optimize performance and tailor the process to their specific needs.

The MATHEMATICA routine can be loaded with

```
In[1]:= Needs["LinApart`"]
```

from any directory, provided the `LinApart.m` file is placed in the standard MATHEMATICA packages directory. Alternatively, one may specify the complete path to the file when loading,

```
In[2]:= Import["/Complete_Path_To_File/LinApart.m"]
```

The main function provided is `LinApart`. The command `LinApart[`*expr, var*`]` returns the partial fraction decomposition of *expr* with respect to the variable *var*, for example,

```
In[3]:= LinApart[1/((1 + x)(2 + x)(3 + x)), x]

Out[3]= 1/(2 (1+x)) - 1/(2+x) + 1/(2 (3+x))
```

Since the algorithm uses the built-in differentiation function, the variable must have the head `Symbol`; if this condition is not fulfilled, an error message is generated and the input is returned unevaluated.

As explained above, `LinApart` assumes that all denominator factors are linear in x. Thus, non-linear denominators, denominators with purely symbolic exponents, as well as non-rational functions of x are by default ignored during decomposition.[4] This implies that for expressions involving non-linear denominators, the output will not be in a completely decomposed form, e.g.,

```
In[4]:= LinApart[x^p Log[x]/((1 + x) (2 + x) (3 + x) (1 + x^2)), x]

Out[4]= x^p Log[x]/(2 (1+x) (1+x^2)) - x^p Log[x]/((2+x) (1+x^2)) + x^p Log[x]/(2 (3+x) (1+x^2))
```

[4] To be more precise, the factors selected for decomposition are required to be linear polynomials in the given variable, i.e., expressions of the form $ax + b$ with a and b independent of x. Thus denominators like $\frac{1}{x+\ln x}$ are also ignored.

In such cases, one can obtain a complete decomposition by writing the non-linear denominators in a fully factorized form. In order to facilitate such manipulations, the option `Factor` is provided. When set to `True`, the input expression is factorized term-by-term before the partial fraction decomposition

```
In[5]:= LinApart[1/((1 - a^2) (1 - x^2)), x, "Factor" -> True]
```

$$\text{Out[5]}= -\frac{1}{2\,(1-a^2)\,(-1+x)} + \frac{1}{2\,(1-a^2)\,(1+x)}$$

Notice that in order to avoid any unnecessary computation, the factorization only affects the variable-dependent part. The factorization is performed using the built-in `Factor` command, i.e., by default the expression is factored over the integers. Hence, factorization will not influence irreducible (over the integers) higher-order denominators, which will continue to be ignored. However, by setting the additional option `GaussianIntegers` to `True` in conjunction with `Factor`, factorization can be extended to allow for constants that are Gaussian integers,

```
In[6]:= LinApart[1/((1 + x) (1 + x^2)), x, "Factor" -> True,
         "GaussianIntegers" -> True]
```

$$\text{Out[6]}= -\frac{\frac{1}{4}+\frac{I}{4}}{-I+x} - \frac{\frac{1}{4}-\frac{I}{4}}{I+x} + \frac{1}{2\,(1+x)}$$

Further application of the built-in `ComplexExpand` command allows the output to be written in a manifestly real form

```
In[7]:= LinApart[1/((1 + x) (1 + x^2)), x, "Factor" -> True,
         "GaussianIntegers" -> True] // ComplexExpand
```

$$\text{Out[7]}= \frac{1}{2\,(1+x)} + \frac{1}{2\,(1+x^2)} - \frac{x}{2\,(1+x^2)}$$

Turning to expressions involving factors with rational exponents, consider the following example

```
In[8]:= LinApart[1/((1 + x)(2 + x)^(1/2)(3 + x)^(1/3+p)), x]
```

$$\text{Out[8]}= \frac{1}{2}\sqrt{2+x}\,(3+x)^{-\frac{1}{3}-p} - \frac{(3+x)^{\frac{2}{3}-p}}{\sqrt{2+x}} + \frac{\sqrt{2+x}\,(3+x)^{\frac{2}{3}-p}}{2\,(1+x)}$$

Notice that the exponents were manipulated as in eq. (16) prior to performing the partial fraction decomposition. In particular, the factors of $\frac{1}{(2+x)^{\frac{1}{2}}}$ and $\frac{1}{(3+x)^{\frac{1}{3}+p}}$ were internally rewritten as

$$\frac{1}{(2+x)^{\frac{1}{2}}} = \frac{1}{2+x}\cdot\sqrt{2+x} \qquad \text{and} \qquad \frac{1}{(3+x)^{\frac{1}{3}+p}} = \frac{1}{3+x}\cdot(3+x)^{\frac{2}{3}-p},$$

with the first factors on the right hand sides entering the partial fraction decomposition. As the example demonstrates, this rewriting is not influenced by the presence of the symbolic constant p in the exponent. In this particular example, the output is in the same form as would be produced by `Apart`.[5]

Finally, we discuss some further options that are designed to optimize the handling of large expressions with many terms. One particular situation which may arise is that the same rational function appears many times in the input with different coefficients. In such cases, rather than applying the partial fraction decomposition routine on individual terms, it can be more advantageous to first gather every unique x-dependent structure. Then, partial fraction decomposition only needs to be performed once per structure. In order to facilitate these operations, the option `PreCollect` is included. When set to `True`, unique x-dependent structures are first gathered in the input, before any partial fraction decomposition takes place. As stated above, when generating the output, the routine distributes the x-independent coefficients over the decomposed expressions, so the final output will not necessarily change when the `PreCollect` option is active. However, the number of internal operations, hence the timing and memory usage, can be significantly different. To demonstrate this, consider, e.g.,

```
In[9]:= expr = 2/((1 + x)(2 + x)) + 1/((1 - a)(1 + x)(2 + x))
         + a/((1 + x)(2 + x)) - a/((1 - a)(1 + x)(2 + x))
         - b/((1 + a)(1 + x)(2 + x)) - (a b)/((1 + a)(1 + x)(2 + x));

In[10]:= LinApart[expr, x]
         AbsoluteTiming[LinApart[expr, x]][[1]]
         MaxMemoryUsed[LinApart[expr, x]]
```

[5] However, exceptions can occur in cases where the exponent is a more elaborate function of rational numbers and symbolic parameters. In this case, what one considers the rational part of the exponent depends on the exact form in which it is written, e.g., consider $(x+a)^{1/2+(1-p)^2}$ and $(x+a)^{3/2-2p+p^2}$ which differ only in that the exponent is expanded in the second form. We choose to extract the rational part without any manipulation of the exponents (e.g., $1/2$ and $3/2$ in the first and second form), which closely matches, but is not exactly identical to the prescription used by `Apart`.

```
Out[10]= 2/(1+x) + 1/((1-a)(1+x)) + a/(1+x) - a/((1-a)(1+x)) - b/((1+a)(1+x))
         - a b/((1+a)(1+x)) - 2/(2+x) - 1/((1-a)(2+x)) - a/(2+x) + a/((1-a)(2+x))
         + b/((1+a)(2+x)) + a b/((1+a)(2+x))

Out[11]= 0.003757

Out[12]= 18016

In[13]:= LinApart[expr, x, "PreCollect" -> True]
         AbsoluteTiming[LinApart[expr, x, "PreCollect" -> True]][[1]]
         MaxMemoryUsed[LinApart[expr, x, "PreCollect" -> True]]

Out[13]= 2/(1+x) + 1/((1-a)(1+x)) + a/(1+x) - a/((1-a)(1+x)) - b/((1+a)(1+x))
         - a b/((1+a)(1+x)) - 2/(2+x) - 1/((1-a)(2+x)) - a/(2+x) + a/((1-a)(2+x))
         + b/((1+a)(2+x)) + a b/((1+a)(2+x))

Out[14]= 0.002276

Out[15]= 15896
```

Indeed, in this simple example, the output with `"PreCollect" -> True` is identical to what is obtained without the `PreCollect` option, however, the timing and memory usage are already somewhat improved. For more elaborate expressions, the improvements can be much more pronounced.

When the `PreCollect` option is set to `True`, one may further specify the option `ApplyAfterPreCollect`, which takes a pure function and applies it to the x-independent coefficients of the unique x-dependent structures which were identified by `PreCollect`. A typical application would be to factor such coefficients, which can be achieved by setting `"ApplyAfterPreCollect" -> Factor`

```
In[16]:= expr = 2/((1 + x)(2 + x)) + 1/((1 - a)(1 + x)(2 + x))
          + a/((1 + x)(2 + x)) - a/((1 - a)(1 + x)(2 + x))
          - b/((1 + a)(1 + x)(2 + x)) - (a b)/((1 + a)(1 + x)(2 + x));

         LinApart[expr, x, "PreCollect" -> True,
          "ApplyAfterPreCollect" -> Factor]

Out[16]= 3/(1+x) + a/(1+x) - b/(1+x) - 3/(2+x) - a/(2+x) + b/(2+x)
```

As the above example shows, this can lead to significant simplifications. However, one must be aware that if the coefficients which arise after gathering unique structures are large, performing the factorization can become quite expensive in terms of runtime. In order to mitigate this to a certain extent, the pure function `GatherByDenominator` is defined. This function gathers its input by denominators and can already lead to substantial simplifications without a complete factoring of the coefficients generated by `PreCollect`,

```
In[17]:= expr = 2/((1 + x)(2 + x)) + 1/((1 - a)(1 + x)(2 + x))
          + a/((1 + x)(2 + x)) - a/((1 - a)(1 + x)(2 + x))
          - b/((1 + a)(1 + x)(2 + x)) - (a b)/((1 + a)(1 + x)(2 + x));

         LinApart[expr, x, "PreCollect" -> True,
          "ApplyAfterPreCollect" -> GatherByDenominator]

Out[17]= 3/(1+x) + a/(1+x) + (-b-a b)/((1+a)(1+x)) - 3/(2+x) - a/(2+x) - (-b-a b)/((1+a)(2+x))
```

In this case, the simplification is not complete, but nevertheless, the output is already in a much simpler form than without the use of the `ApplyAfterPreCollect` option. It is important to note though, that the choice of the appropriate `ApplyAfterPreCollect` setting is highly context-dependent and may require careful consideration of the specific characteristics of the algebraic expressions involved. The list of options is summarized in Table 1.

3.2. The C routine

We have also implemented our routine based on eqs. (11) and (15) in the C programming language for rational functions of the form

$$x^l \prod_{k=1}^{n} \frac{1}{(x-a_k)^{m_k}}, \tag{17}$$

where l is a non-negative integer and m_k, $k = 1, \ldots, n$ are strictly positive integers. Moreover, the a_k are assumed to be distinct. It is packaged into a standalone executable as well as a library suitable for linking with other software. In particular we provide the header file `LinApart.h` which can be used as a developer library to create other tools and programs that require partial fraction decomposition functionality.

Table 1
The list of options for the `LinApart` command.

Option	Description
`Factor`	`True/False`: If set to `True`, the input expression is factorized term-by-term before partial fraction decomposition. The default value is `False`.
`GaussianInteger`	`True/False`: If set to `True` in conjunction with `Factor`, factorization of the input expression is performed over the Gaussian integers. The default value is `False`.
`PreCollect`	`True/False`: If set to `True`, unique variable-dependent structures are gathered in the input before partial fraction decomposition. The default value is `False`.
`ApplyAfterPreCollect`	pure function (e.g., `Factor`): If the option `PreCollect` is set to `True`, the pure function specified by this option is applied to the coefficients of the unique variable-dependent structures identified by `PreCollect`. Typical functions might be `Factor` or `GatherByDenominator`. By default, this option is empty (no function is applied to the coefficients).

One of the key features of our implementation is its efficiency in terms of both time and space complexity. The GNU Multiple Precision Arithmetic Library (GMP) [24] library is used for all multiple-precision integer operations, such as calculations of the factorials of large numbers, while dynamic memory allocation is carefully managed. To limit memory usage, the program employs a buffering strategy where intermediate results are accumulated in a fixed-size buffer. When the buffer reaches its capacity, the contents are flushed to disk. This approach allows the program to handle very large inputs that would otherwise exceed available memory. Only the memory required by GMP to perform the arithmetic operations is consumed such that the memory usage will not exceed a couple kilobytes in all but the most extreme applications.

The standalone program can be compiled on Unix-like systems simply by running `make` in the `LinApart/C` directory, as the only external dependency (apart from the C compiler) is the GMP library which is widely available. The general usage of the program is as follows:

```
./LinApart <exponents> <roots> [-q] [-o OutputFileName]
```

where `<exponents>` should contain the comma separated list of the $(m+1)$ integers $l, m_1, \dots m_n$ defined in eq. (17). I.e., the first entry represents the exponent of the monomial in the numerator and the rest are the multiplicities of the factors in the denominator. `<roots>` should contain the comma separated list of roots $a_1, \dots, a_n$. The decomposition variable is automatically set to x. For instance, the command:

```
./LinApart "1,2,1" "a,b"
```

will perform the partial fraction decomposition of

$$\frac{x}{(x-a)^2(x-b)},$$

returning the following output string

```
((a)-(b))^(-2)*(-1)*1*1*(a)^1*(x-(a))^(-1)*1+((a)-(b))^(-1)
*(1)*1*1*(a)^1*(x-(a))^(-2)*1+((a)-(b))^(-1)*(1)*1*1
*(x-(a))^(-1)*1+((b)-(a))^(-2)*(1)*1*1*(b)^1*(x-(b))^(-1)*1+0
```

Note that the output may contain factors and exponents of 1 and terms that equal 0. These are not errors but result from the program's internal mechanisms designed to maintain high performance. These superfluous elements will typically be simplified away by symbolic manipulation programs that use this output as input. The program generates a formal output for all inputs, but it assumes that all roots are distinct. If repeated roots are provided, the output will contain undefined expressions.

Finally, by default, the result will be written to the standard output and saved in a file named `LA_result.out`. To improve execution speed while decomposing very large expressions, writing to the standard output can be suppressed by using the `-q` flag. Furthermore, users can specify their preferred filename with the `-o` flag. These usage instructions can also be accessed with the command:

```
./LinApart -h
```

4. Performance

As stated in the Introduction, the motivation for developing `LinApart` was to obtain a highly efficient univariate partial fraction decomposition algorithm. In this section, we examine the performance of our implementation with respect to the standard `Apart` command of MATHEMATICA as a function of the complexity of the input. First, let us consider what makes a rational fraction "complex". Several factors come to mind:

1. The number of distinct denominator factors.
2. The complexity of each individual denominator. In fact, even considering only linear denominators of the form $x - a_i$, the roots a_i may be functions of further variables and symbolic constants.
3. The multiplicity of the denominator factors.
4. The polynomial order of the numerator.

These various aspects of complexity can be captured by choosing suitable input functions for the partial fraction decomposition routines. For example, consider the expression

$$\prod_{i=1}^{n} \frac{1}{x - P_i^{(k)}(y)} \qquad \text{with} \qquad P_i^{(k)}(y) = \sum_{j=1}^{k} a_{i,j} y^j \,. \tag{18}$$

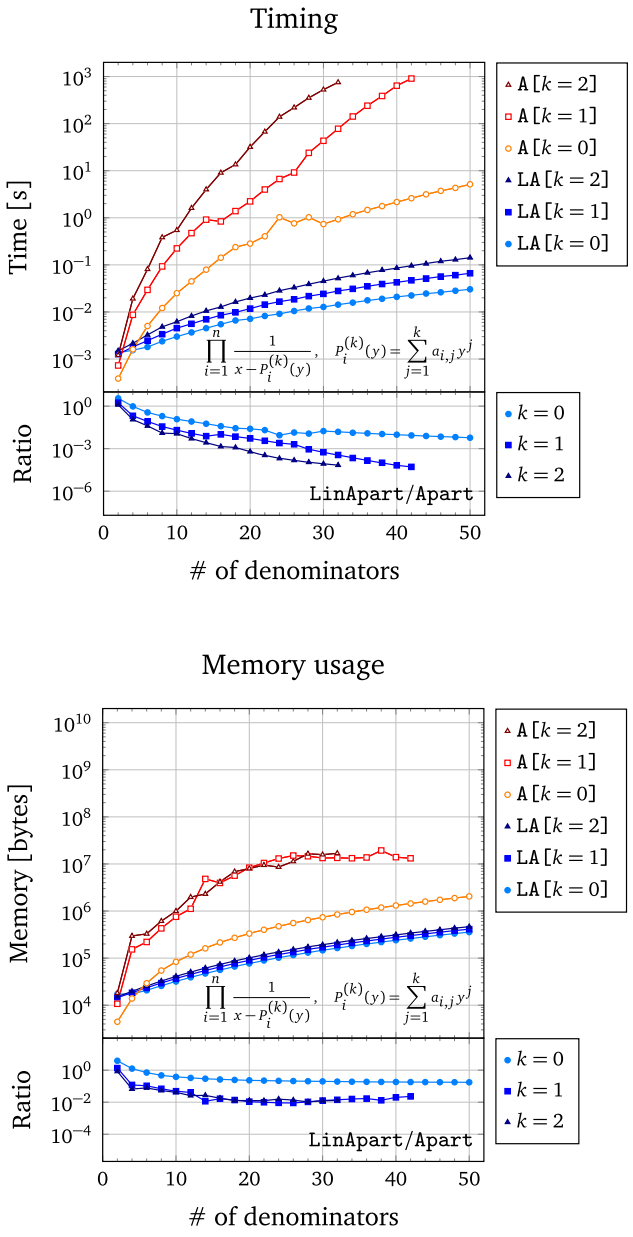


Fig. 1. Timings and memory usage of `Apart` and `LinApart` (denoted as `A` and `LA` in the legend) on rational functions of x with n distinct denominators of multiplicity one of the form $x - P^{(k)}(y)$. The roots $P^{(k)}(y)$ are chosen to be symbolic polynomials in the auxiliary variable y of order k. Various curves correspond to different polynomial orders $k = 0, 1, 2$ in the roots. The numerator has been set to 1.

Choosing the $a_{i,j}$ to be symbolic constants, we can vary the complexity by varying the total number of factors n, as well as the polynomial order k of the roots. In Fig. 1, we present the timings and memory usage[6] of `Apart` and `LinApart` on expressions of the form given in eq. (18). During each evaluation, we have constrained the runtime in MATHEMATICA 12.0 to a maximum of 10^3 seconds. Hence, missing data points represent computations that did not finish with this time constraint.[7] The number of denominators was chosen between 2 and 50 and we varied the polynomial order of the roots between 0 and 2. The figure clearly shows that for most of the considered cases, `LinApart` outperforms `Apart` both in terms of timing and memory usage. Indeed, already for just six denominators, `LinApart` produces a speedup of a factor between ∼2 and ∼20 depending on the polynomial order of the roots. For ten denominators, the speedup is between a factor of ∼5 and ∼100, while for tens of denominators and roots of polynomial order two, we observe speedups of factors greater than 10^4. On these expressions, `LinApart` also typically outperforms `Apart` in terms of memory used by factors of ∼10–100, depending on the number of denominators and the complexity of the roots.

Next, let us study the effect of higher multiplicities. To begin, we examine the case when only one out of the n distinct denominators has multiplicity $m > 1$,

[6] Timing and memory usage information were obtained with the `AbsoluteTiming` and `MaxMemoryUsed` commands.
[7] We have performed these evaluations on a desktop PC with an Intel Core i5 processor and 16Gb of RAM.

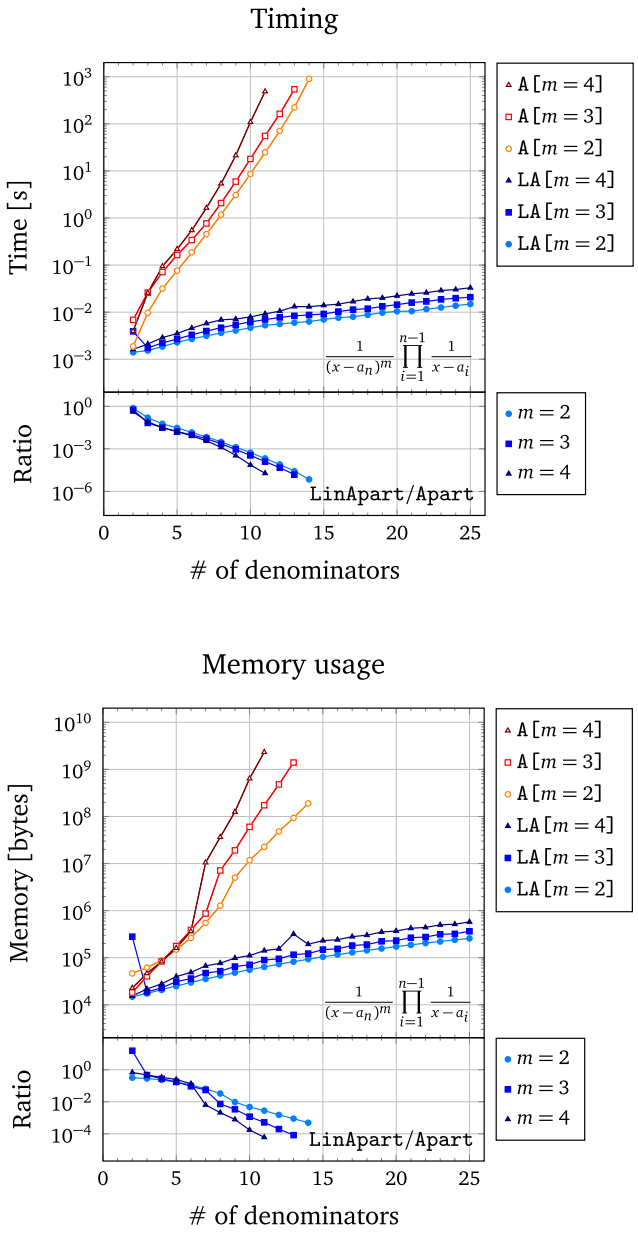


Fig. 2. Timings and memory usage of `Apart` and `LinApart` (denoted as `A` and `LA` in the legend) on rational functions of x with n distinct denominators of multiplicity one, except for the last denominator, whose multiplicity is m. The roots a_i are chosen to be symbolic constants. Various curves correspond to different multiplicities $m = 2, 3, 4$ of the last denominator. The numerator has been set to 1.

$$\frac{1}{(x-a_n)^m}\prod_{i=1}^{n-1}\frac{1}{x-a_i}\,, \qquad 2 \le m\,. \tag{19}$$

The results for timings and memory usage of `Apart` and `LinApart` on inputs of the form of eq. (19) are presented in Fig. 2. As before, a time constraint of 10^3 seconds was imposed on every evaluation. The number of denominators was varied between 2 and 25, and the multiplicity of the last denominator between 2 and 4. Again, we observe a dramatic improvement in performance using `LinApart`, both in terms of necessary time and memory. In fact, already for expressions with only three denominators, we obtain a speedup of a factor of ~ 10. On expressions with ten denominators, the speedup is already greater than a factor of 10^3. Increasing the number of denominators further, we quickly exhaust the cases where the evaluation with `Apart` finishes in the chosen time constraint. Indeed, for $m = 2$ this happens already at $n = 14$ (i.e., just 13 denominators of multiplicity one and a single denominator with multiplicity two). In this case, we observe a speedup of a factor greater than 10^5. Examining the memory usage, we also see significant gains of up to factors of 10^4 in favour of `LinApart`.

It is also interesting to consider the case of multiple denominators whose multiplicity is greater than one. Thus, in Fig. 3, we examine the timings and memory usage of `Apart` and `LinApart` on expressions of the form

$$\prod_{i=1}^{n-l}\frac{1}{x-a_i}\prod_{j=n-l+1}^{n}\frac{1}{(x-a_j)^2}\,, \qquad 1 \le l \le n\,, \tag{20}$$

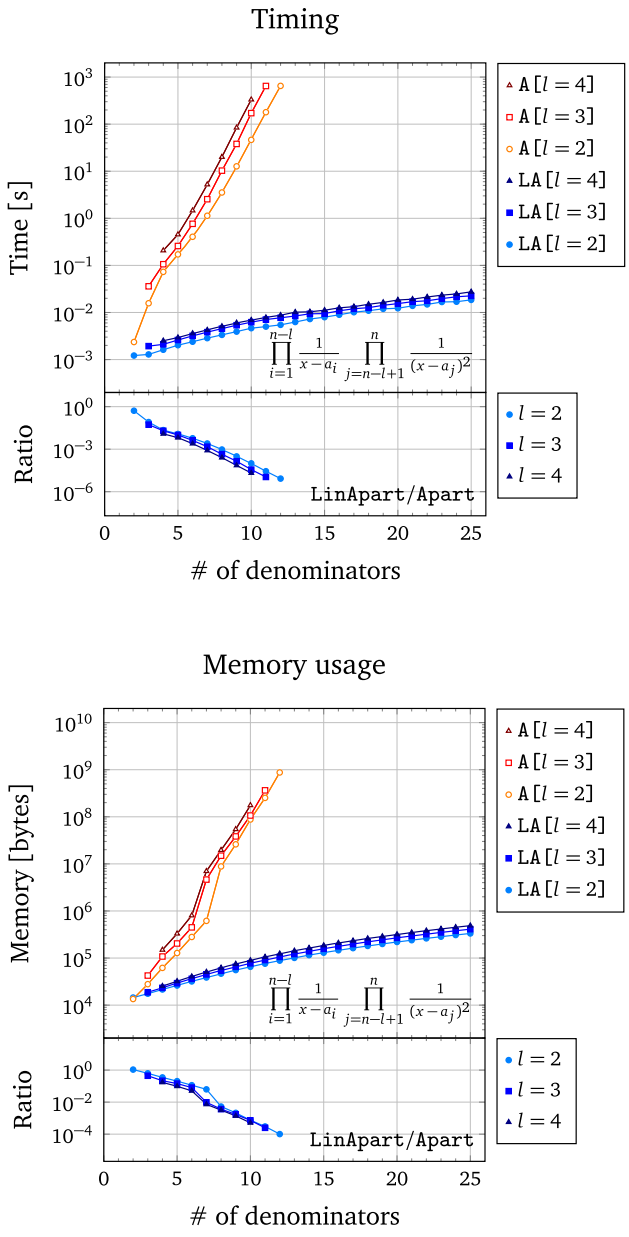


Fig. 3. Timings and memory usage of `Apart` and `LinApart` (denoted as `A` and `LA` in the legend) on rational functions of x with n distinct denominators. The multiplicities of the first $n-l$ denominators are one, while the last l denominators have multiplicity two. The roots a_i are chosen to be symbolic constants. Various curves correspond to the different number $l = 2, 3, 4$ of quadratic denominators. The numerator has been set to 1.

i.e., l out of the n distinct denominators have multiplicity two, while the rest of the $(n-l)$ denominators have multiplicity one. In terms of both evaluation times and memory used, we observe the same dramatic performance gains using `LinApart` as in the previous case of a single denominator of higher multiplicity. Again, speedups of factors of $\sim 10^4$–10^5 are obtained for expressions with a total number of only around ten denominators. At the same time, the required memory also decreases by up to four orders of magnitude.

Next, we consider the effects of including non-trivial numerators,

$$P^{(k)}(x) \prod_{i=1}^{n} \frac{1}{x - a_i} \qquad \text{with} \qquad P^{(k)}(x) = \sum_{j=1}^{k} b_j x^j \,. \tag{21}$$

The timings and memory usage obtained with `Apart` and `LinApart` on improper rational function inputs of the form of eq. (21) are presented in Fig. 4 for numerators with polynomial orders $k = n$, $\lfloor 3n/2 \rfloor$ and $2n$. Once again, huge improvements in performance of `LinApart` over `Apart` are apparent in both timing and memory used. Concerning memory usage in particular, we note that during these evaluations, we had to enforce a memory constraint of 12Gb in order to avoid the automatic closing of the MATHEMATICA kernel while `Apart` was running.

For proper rational functions, we present the timings and memory usage of `Apart` and `LinApart` in Fig. 5. Here the total number of denominators was varied between 2 and 50 and we have chosen numerators with polynomial orders $k = 1$, 2 and 3. As in all previous cases `LinApart` outperforms `Apart` except for the simplest cases, especially in terms of timing. Interestingly, in this case only, we do not observe very large (i.e., orders of magnitude) gains in terms of memory usage in favour of `LinApart`.

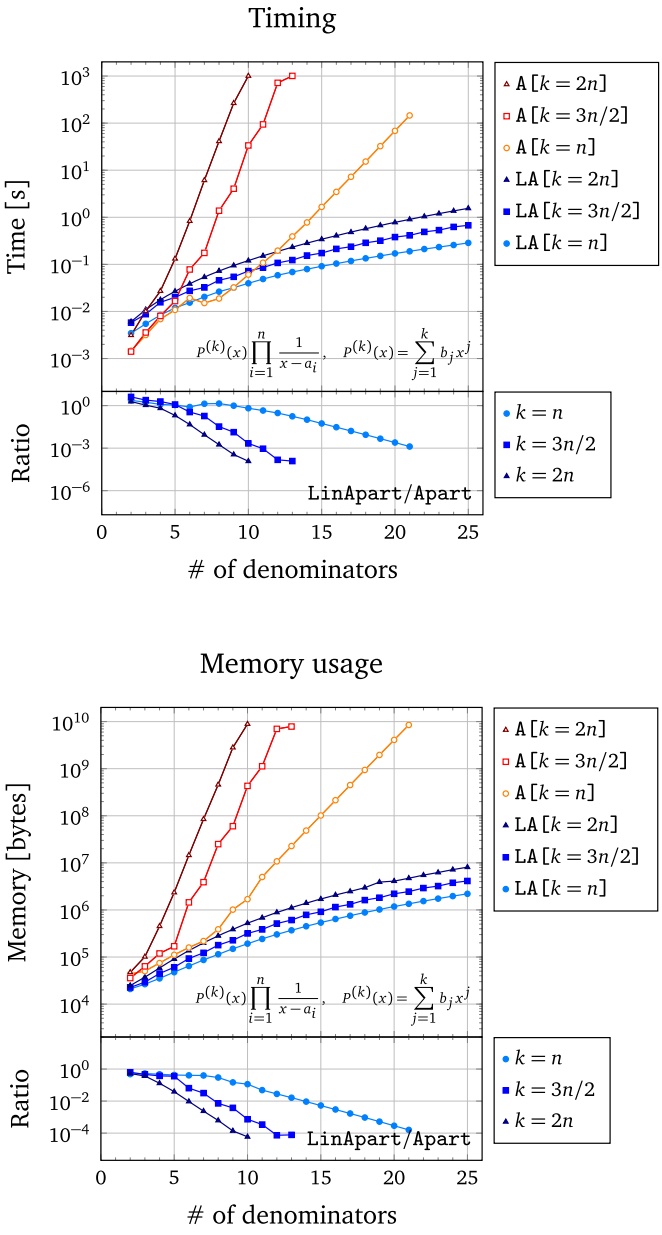


Fig. 4. Timings and memory usage of `Apart` and `LinApart` (denoted as `A` and `LA` in the legend) on improper rational functions of x with n distinct denominators of multiplicity one. The roots a_i are chosen to be symbolic constants, while the numerator is a symbolic polynomial $P^{(k)}$ of x. Various curves correspond to different polynomial orders $k = n, \lfloor 3n/2 \rfloor, 2n$.

To present a concrete application, we show an example which emerges during the analytic computation of phase-space integrals relevant for setting up a local subtraction scheme beyond next-to-leading order. After choosing a particular phase-space parametrization we encounter expressions of the form

$$
\begin{aligned}
f(x_a, x_b; y) = \Big[& (-4+y)(1-y+x_b y)(2-y+x_b y)(4-y+x_b y)(1-x_a-y+x_b y)^3 \\
& \times (-1+x_a-y+x_b y)(-4-4x_b-y+x_b y)(-4x_b-y+x_b y) \\
& \times (-4x_a-4x_b-y+x_b y)(4x_a-4x_b-y+x_b y)(2+2x_b-y+x_b y)^3 \\
& \times (6+2x_b-y+x_b y)(2-4x_a+2x_b-y+x_b y)(2+4x_a+2x_b-y+x_b y) \\
& \times (-1+x_a-x_a y+x_a x_b y)(1+x_a-x_a y+x_a x_b y)(-2+2x_a-x_a y+x_a x_b y) \\
& \times (2+2x_a-x_a y+x_a x_b y)(-x_b+x_a x_b-x_a y+x_a x_b y)^3 \\
& \times (-4+2x_a+2x_a x_b-x_a y+x_a x_b y)(4+2x_a+2x_a x_b-x_a y+x_a x_b y) \\
& \times (1-2x_a+x_a^2-y-x_a y+x_b y+x_a x_b y) \\
& \times \left(2x_b-2x_a x_b+x_a y-x_b y-x_a x_b y+x_b^2 y\right)^3 \Big]^{-1},
\end{aligned}
\tag{22}
$$

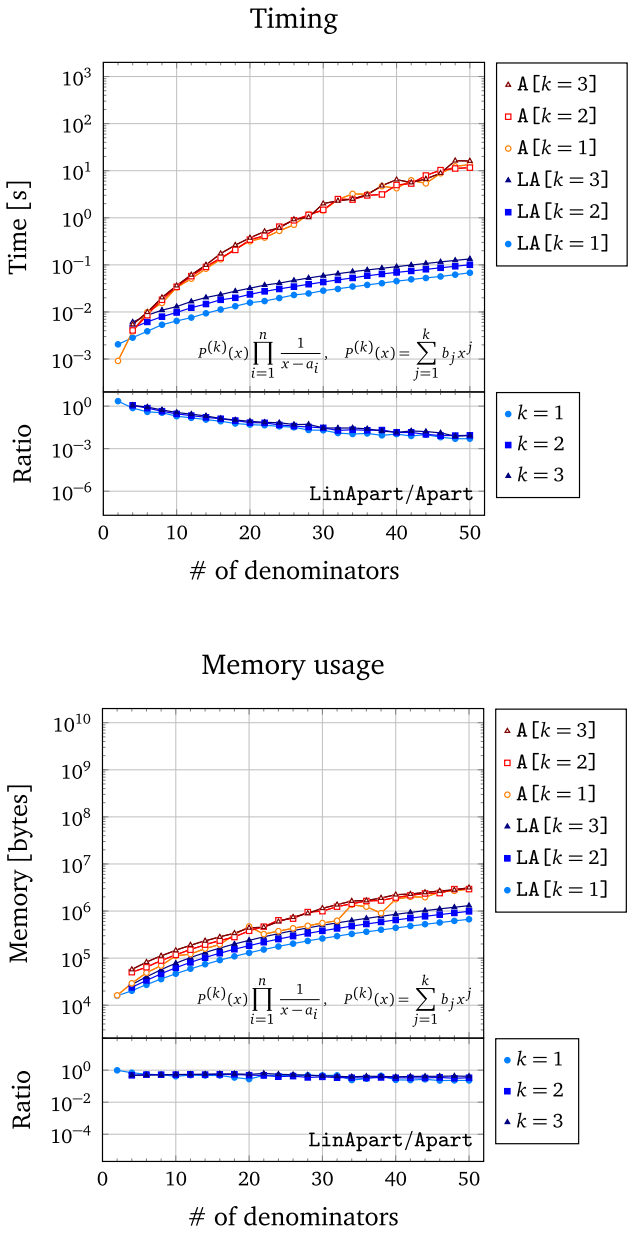


Fig. 5. Timings and memory usage of `Apart` and `LinApart` (denoted as `A` and `LA` in the legend) on proper rational functions of x with n distinct denominators of multiplicity one. The roots a_i are chosen to be symbolic constants, while the numerator is a symbolic polynomial $P^{(k)}$ of x. Various curves correspond to different polynomial orders $k = 1, 2, 3$.

which must be integrated symbolically over y. To do so, one must first perform the partial fraction decomposition with respect to y. The rational function in eq. (22) has 23 linear (in y) denominators, 4 of which have multiplicity 3. A quick glance at Fig. 3 shows that `Apart` already requires close to 10^3 seconds to decompose a rational function with just 10 denominators, 4 of which have multiplicity 2. A naive extrapolation to 23 denominators then puts the time required for the decomposition of $f(x_a, x_b; y)$ at the order of 10^9 seconds.[8] On the other hand, `LinApart` performs the partial fraction decomposition in a mere $\sim 10^{-2}$ seconds (without using any of the provided options). As expressions actually encountered in real computations typically have hundreds or even thousands of rational functions of the type in eq. (22), this is a vital improvement.

We finish this section by offering some comments on why we believe our method to be so efficient, especially on large symbolic expressions. To our knowledge, the current best complexity bound for the univariate partial fraction decomposition of proper fractions is $\mathcal{O}(n \log^2 n)$, where n is the degree of the denominator [13]. The timings for `LinApart` presented in the plots above are indeed consistent with this theoretical scaling. Hence, naively we would expect only a constant factor improvement over other methods for large n. Nonetheless, our analysis clearly demonstrates that the implementation of `Apart` in MATHEMATICA does not follow this scaling at all. In fact, in certain situations, the behaviour looks exponential, see

[8] As we are dealing now with denominators of multiplicity 1 and 3 instead of 1 and 2 as in Fig. 3, we expect this number to be a rough lower limit. The situation is made even worse by the fact that eq. (22) contains roots which are non-trivial polynomials of x_a and x_b. This also has a significant influence on the efficiency of `Apart`. Hence we expect the actual time required to be several orders of magnitude larger than this naive estimate.

Figs. 2–4 in particular. However, the scaling is drastically improved if the roots are numeric rather than symbolic, which offers a possible explanation for our observations.

One common way of performing univariate partial fraction decomposition is to employ the Extended Euclidean Algorithm to solve $A_i(x)Q_i(x) + A_j(x)Q_j(x) = 1$ for $A_i(x)$ and $A_j(x)$, given the coprime denominators $Q_i(x)$ and $Q_j(x)$ [13]. This algorithm is well-defined for univariate polynomials with coefficients in some field. However, when using symbolic roots, all operations on the field elements must now be performed on these symbolic expressions. In particular if the roots are symbolic polynomials in some variables other than x, these operations will include expensive computations on multivariate rational expressions, such as finding greatest common divisors. On the other hand, by performing this polynomial arithmetic "analytically", `LinApart` avoids most of this complexity.

5. Conclusions and outlook

In this paper, we have introduced the `LinApart` routine, which provides a fast and efficient implementation of univariate partial fraction decomposition for rational functions with fully factorized denominators. Our implementation is based on a simple closed formula for the decomposed function and we provide realizations in the WOLFRAM MATHEMATICA and C languages. The MATHEMATICA routine leverages the highly-efficient built-in differentiation routine, while our C code can be interfaced with computer algebra systems such as FORM, where symbolic differentiation is not available.

Concentrating on the MATHEMATICA implementation, we have found that `LinApart` outperforms the built-in `Apart` function on virtually any input expression both in terms of timing and memory used. The increase of efficiency is dramatic already for reasonably small expressions, especially if the multiplicities of some of the denominator factors are greater than one. Indeed, for rational functions involving only around ten independent denominators, with only a few (say two to four) denominator factors of multiplicity two, speedups of up to five orders of magnitude are observed. At the same time, the required memory decreases by up to four orders of magnitude. Thus, `LinApart` is able to efficiently handle decomposition problems that are intractable for the built-in routines.

A limitation of the current implementation concerns the treatment of non-linear (in the decomposition variable) denominator factors. While any polynomial can be factored into linear factors over the complex numbers, the partial fraction decomposition problem over the reals is understood to allow the appearance of quadratic denominators as well. Including such factors in our algorithm is in principle possible, but finding the most efficient way to do so is non-trivial and is left for future work.

CRediT authorship contribution statement

B. Chargeishvili: Writing – original draft, Validation, Software, Investigation, Conceptualization. **L. Fekésházy:** Writing – original draft, Validation, Software, Investigation, Formal analysis, Conceptualization. **G. Somogyi:** Writing – review & editing, Writing – original draft, Visualization, Supervision, Conceptualization. **S. Van Thurenhout:** Writing – review & editing, Writing – original draft, Validation, Software, Investigation, Formal analysis, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

The authors would like to thank Sven-Olaf Moch for useful discussions and Vitaly Magerya for his comments on the manuscript. This work has been supported by grant K143451 of the National Research, Development and Innovation Fund in Hungary. Furthermore, the work of G.S. was supported by the Bolyai Fellowship program of the Hungarian Academy of Sciences. The work of L.F. was supported by the German Academic Exchange Service (DAAD) through its Bi-Nationally Supervised Scholarship program.

Data availability

No data was used for the research described in the article.

References

[1] A.B. Goncharov, Multiple polylogarithms, cyclotomy and modular complexes, Math. Res. Lett. 5 (1998) 497–516, https://doi.org/10.4310/MRL.1998.v5.n4.a7, arXiv:1105.2076.
[2] E. Remiddi, J.A.M. Vermaseren, Harmonic polylogarithms, Int. J. Mod. Phys. A 15 (2000) 725–754, https://doi.org/10.1142/S0217751X00000367, arXiv:hep-ph/9905237.
[3] A.B. Goncharov, Multiple polylogarithms and mixed Tate motives, arXiv:math/0103059, 2001.
[4] C. Anastasiou, C. Duhr, F. Dulat, B. Mistlberger, Soft triple-real radiation for Higgs production at N3LO, J. High Energy Phys. 07 (2013) 003, https://doi.org/10.1007/JHEP07(2013)003, arXiv:1302.4379.
[5] C. Duhr, Mathematical aspects of scattering amplitudes, in: Theoretical Advanced Study Institute in Elementary Particle Physics: Journeys Through the Precision Frontier: Amplitudes for Colliders, 2015, pp. 419–476, arXiv:1411.7538.
[6] F. Feng, `$Apart`: a generalized mathematica apart function, Comput. Phys. Commun. 183 (2012) 2158–2164, https://doi.org/10.1016/j.cpc.2012.03.025, arXiv:1204.2314.
[7] D. Bendle, J. Böhm, W. Decker, A. Georgoudis, F.-J. Pfreundt, M. Rahn, P. Wasser, Y. Zhang, Integration-by-parts reductions of Feynman integrals using singular and GPI-space, J. High Energy Phys. 02 (2020) 079, https://doi.org/10.1007/JHEP02(2020)079, arXiv:1908.04301.
[8] J. Boehm, M. Wittmann, Z. Wu, Y. Xu, Y. Zhang, IBP reduction coefficients made simple, J. High Energy Phys. 12 (2020) 054, https://doi.org/10.1007/JHEP12(2020)054, arXiv:2008.13194.
[9] S. Badger, C. Brønnum-Hansen, D. Chicherin, T. Gehrmann, H.B. Hartanto, J. Henn, M. Marcoli, R. Moodie, T. Peraro, S. Zoia, Virtual QCD corrections to gluon-initiated diphoton plus jet production at hadron colliders, J. High Energy Phys. 11 (2021) 083, https://doi.org/10.1007/JHEP11(2021)083, arXiv:2106.08664.
[10] S. Badger, H.B. Hartanto, S. Zoia, Two-loop QCD corrections to $Wb\bar{b}^-$ production at hadron colliders, Phys. Rev. Lett. 127 (1) (2021) 012001, https://doi.org/10.1103/PhysRevLett.127.012001, arXiv:2102.02516.

[11] M. Heller, A. von Manteuffel, MultivariateApart: generalized partial fractions, Comput. Phys. Commun. 271 (2022) 108174, https://doi.org/10.1016/j.cpc.2021.108174, arXiv:2101.08283.
[12] H. Strubbe, Manual for schoonschip: a CDC 6000 / 7000 program for symbolic evaluation of algebraic expressions, Comput. Phys. Commun. 8 (1974) 1–30, https://doi.org/10.1016/0010-4655(74)90081-2.
[13] H.T. Kung, D.M. Tong, Fast algorithms for partial fraction decomposition, SIAM J. Comput. 6 (3) (1977) 582–593, https://doi.org/10.1137/0206042.
[14] P. Wang, A p-Adic Algorithm for Univariate Partial Fractions, 1981, pp. 212–217.
[15] J. Mahoney, B. Sivazlian, Partial fractions expansion: a review of computational methodology and efficiency, J. Comput. Appl. Math. 9 (3) (1983) 247–269, https://doi.org/10.1016/0377-0427(83)90018-3.
[16] J. von zur Gathen, J. Gerhard, Modern Computer Algebra, 3rd edition, Cambridge University Press, 2013.
[17] Y. Kim, B. Lee, Partial fraction decomposition by repeated synthetic division, Am. J. Comput. Math. 06 (2016) 153–158, https://doi.org/10.4236/ajcm.2016.62016.
[18] G. Somogyi, Z. Trocsanyi, A subtraction scheme for computing QCD jet cross sections at NNLO: integrating the subtraction terms. I, J. High Energy Phys. 08 (2008) 042, https://doi.org/10.1088/1126-6708/2008/08/042, arXiv:0807.0509.
[19] P. Bolzoni, G. Somogyi, Z. Trocsanyi, A subtraction scheme for computing QCD jet cross sections at NNLO: integrating the iterated singly-unresolved subtraction terms, J. High Energy Phys. 01 (2011) 059, https://doi.org/10.1007/JHEP01(2011)059, arXiv:1011.1909.
[20] G. Somogyi, A subtraction scheme for computing QCD jet cross sections at NNLO: integrating the doubly unresolved subtraction terms, J. High Energy Phys. 04 (2013) 010, https://doi.org/10.1007/JHEP04(2013)010, arXiv:1301.3919.
[21] V. Del Duca, G. Somogyi, Z. Trocsanyi, Integration of collinear-type doubly unresolved counterterms in NNLO jet cross sections, J. High Energy Phys. 06 (2013) 079, https://doi.org/10.1007/JHEP06(2013)079, arXiv:1301.3504.
[22] J.A.M. Vermaseren, New features of FORM, arXiv:math-ph/0010025, 2000.
[23] J. Kuipers, T. Ueda, J.A.M. Vermaseren, J. Vollinga, FORM version 4.0, Comput. Phys. Commun. 184 (2013) 1453–1467, https://doi.org/10.1016/j.cpc.2012.12.028, arXiv:1203.6543.
[24] GMP: The GNU Multiple Precision Arithmetic Library, https://gmplib.org/. (Accessed 1 May 2024).