DESY-16-219

Fuchsia: a tool for reducing differential equations for Feynman master integrals to epsilon form

Version 16.11.14

Oleksandr Gituliar*[a] and Vitaly Magerya[b]

[a] *II. Institut für Theoretische Physik, Universität Hamburg, Luruper Chaussee 149, D-22761 Hamburg, Germany*
[b] *Novomoskovsk, Ukarine*

January 16, 2017

Abstract

We present Fuchsia — an implementation of the Lee algorithm, which for a given system of ordinary differential equations with rational coefficients $\partial_x \mathbf{f}(x,\epsilon) = \mathbb{A}(x,\epsilon)\,\mathbf{f}(x,\epsilon)$ finds a basis transformation $\mathbb{T}(x,\epsilon)$, i.e., $\mathbf{f}(x,\epsilon) = \mathbb{T}(x,\epsilon)\,\mathbf{g}(x,\epsilon)$, such that the system turns into the *epsilon form*: $\partial_x \mathbf{g}(x,\epsilon) = \epsilon\,\mathbb{S}(x)\,\mathbf{g}(x,\epsilon)$, where $\mathbb{S}(x)$ is a Fuchsian matrix. A system of this form can be trivially solved in terms of polylogarithms as a Laurent series in the dimensional regulator ϵ. That makes the construction of the transformation $\mathbb{T}(x,\epsilon)$ crucial for obtaining solutions of the initial system.

In principle, Fuchsia can deal with any regular systems, however its primary task is to reduce differential equations for Feynman master integrals. It ensures that solutions contain only regular singularities due to the properties of Feynman integrals.

*Corresponding author; email address: oleksandr.gituliar@desy.de

PROGRAM SUMMARY

Program Title: Fuchsia

Authors: O. Gituliar and V. Magerya

Program obtainable from: https://github.com/gituliar/fuchsia/

Journal Reference:

Catalog identifier:

Licensing provisions: ISC license

Programming language: Python 2.7

Operating system: Linux, Unix-like

RAM: Dependent upon the input data. Expect hundreds of megabytes.

Keywords: Computer algebra, Feynman integrals, differential equations, epsilon form, Fuchsian form, Moser reduction

Classification: 5 Computer Algebra, 11.1 High Energy Physics and Computing

External routines/libraries: SageMath (7.0 or higher), Maple (optional)

Nature of problem: Feynman master integrals may be calculated from solutions of a linear system of differential equations with rational coefficients. Such a system can be easily solved as an ϵ-series when its epsilon form is known. Hence, a tool which is able to find the epsilon form transformations can be used to evaluate Feynman master integrals.

Solution method: Methods of Moser [Mos59] and Lee [Lee15].

Restrictions: Systems of single-variable differential equations are considered, however unknown functions may also depend on other symbolic arguments. A system needs to be reducible to Fuchsian form and eigenvalues of its residues must be of the form $n + m\,\epsilon$, where n is integer.

Running time: Depends upon the input data, its size and complexity. Around an hour in total for an example 74×74 matrix with 20 singular points on a PC with a 1.7GHz Intel Core i5 CPU.

Contents

1 Introduction

More than 60 years have passed since Richard Feynman proposed a diagrammatic approach for calculating perturbative processes in quantum field theories. Since then Feynman integrals calculus has grown to a separate branch of mathematical physics with a big community of scientists conducting research in this exciting field. With no doubt we can say that none of the recent discoveries in the high-energy particle physics could happen without precise theoretical calculations, which are based on the Feynman integrals calculation techniques. It is also clear that such techniques will play a key role for discoveries at the present and future high-energy colliders, hence their development and further improvement are a very important task.

Recent progress in computational techniques has made it possible to automate the calculation of Feynman integrals; problems which seemed impossible 10 years ago now are successfully solved with state-of-the-art computer algorithms. Among the most popular are integration-by-parts (IBP) reduction [CT81] and the method of differential equations (DE) [Kot91c, Kot91b, Kot91a]; for a detailed overview of these and other methods see [Smi06].

In this paper we discuss the method of differential equations. In particular, we focus on the fact that a solution to the system of DEs may be easily found as an ϵ-series when an epsilon form of this system is known [Hen13]. We consider a general algorithm to find an epsilon form of a given system of differential equations in one variable developed by Lee [Lee15], for the review of the subject see [Hen15] and [Pap14, Tan15, ABB+16]. The Lee algorithm allows

1. To find a Fuchsian form of the system using improved Moser reduction algorithm [Mos59]; and then

2. To normalize eigenvalues of the Fuchsian system in all singular points.

If these two steps are successfully completed[1] then a transformation which puts the initial system into the epsilon form may be easily found. Although this method is focused on systems for Feynman integrals, it also may be successfully used for different problems provided that the requirements on particular properties of the initial system are satisfied. For another method for finding the epsilon form of a given system with multiple scales see [Mey16a, Mey16b].

Until recently, no implementation of the Lee method was made publicly available. This fact motivated us to develop `Fuchsia` — the first public implementation of the Lee algorithm [Lee15] which was presented in [GM16]. Another implementation of this method, called `Epsilon`, was recently presented in [Pra17]. In this paper we provide a description of some implementation and usage details of `Fuchsia` which together with algorithms and tools for the integration-by-parts reduction [Lap00, Smi08, vMS12, Lee12, Lee14, SS13, Smi15, GLZ16] form a powerful tandem for evaluating Feynman integrals.

This paper is organized as follows: in Section 2 we introduce notation and definitions followed by a brief review of the Lee method and related algorithms implemented in `Fuchsia`. In Section 3 we describe how to install and how to use `Fuchsia` from different environments, depending on your goal and programming experience.

2 Overview of the Lee method

2.1 Notation and definitions

Let us consider a system of ordinary differential equations (ODEs) of this form:

$$\partial_x \mathbf{J}(x,\epsilon) = \mathbb{A}(x,\epsilon)\,\mathbf{J}, \tag{1}$$

where $\mathbf{J}(x,\epsilon)$ is a column-vector of n unknown functions (e.g., master integrals); x is a free variable; ϵ is an infinitesimally small parameter (e.g., a dimensional regulator in $d = 4 - 2\epsilon$ dimensions); $\mathbb{A}(x,\epsilon)$ is an $n \times n$ matrix, rational in both x and ϵ.

In the general case $\mathbb{A}(x,\epsilon)$ may have a finite number of poles in x at $x \in \{x_k\}$, including a pole at infinity. The asymptotic behavior of $\mathbb{A}(x,\epsilon)$ around these poles can be described as:

$$\mathbb{A}(x \to x_k, \epsilon) = \begin{cases} (\mathbb{A}_{k0}(\epsilon) + \mathbb{A}_{k1}(\epsilon)\,(x - x_k) + \dots)/(x - x_k)^{1+p_k} & \text{if } x_k \neq \infty, \\ -(\mathbb{A}_{k0}(\epsilon) + \mathbb{A}_{k1}(\epsilon)\,x^{-1} + \dots)\,x^{-1+p_k} & \text{if } x_k = \infty, \end{cases} \tag{2}$$

where p_k is the *Poincaré rank* of $\mathbb{A}(x,\epsilon)$ at the singular point $x = x_k$. If $p_k = 0$, we call $\mathbb{A}(x,\epsilon)$ *Fuchsian in* $x = x_k$, and $\mathbb{A}_{k0}$ *matrix residue* of $\mathbb{A}(x,\epsilon)$ at $x = x_k$. If all $p_k = 0$, we call $\mathbb{A}(x,\epsilon)$ *Fuchsian*.

[1] In principle, the first step can always be done because Feynman integrals contain only logarithmic singularities. For the discussion of potential complications in the second step see Section 2.

The behavior of the system at $x = \infty$ is a bit of a special case, but it is essential in the overall reduction process. One must always keep in mind the $x = \infty$ point, and treat it on the same footing as other singular points.

Equivalent systems. We are interested in transforming system (1) into a simpler form. For this purpose let us consider a change of basis from $\mathbf{J}$ into $\mathbf{J}'$ using the linear transformation $\mathbb{T}(x, \epsilon)$:

$$\mathbf{J} = \mathbb{T}(x, \epsilon)\, \mathbf{J}'. \tag{3}$$

This leads to the *equivalent system* of ODEs

$$\partial_x \mathbf{J}' = \mathbb{A}'(x, \epsilon)\, \mathbf{J}', \tag{4}$$

with the new matrix being

$$\mathbb{A}'(x, \epsilon) = \mathbb{T}^{-1} \left(\mathbb{A}\mathbb{T} - \partial_x \mathbb{T} \right). \tag{5}$$

Generally speaking, the transformations $\mathbb{T}(x, \epsilon)$ may have an arbitrary form. However in the scope of this paper we will require the transformation matrices to be rational in both x and ϵ. This restriction guarantees that the equivalent matrix $\mathbb{A}'(x, \epsilon)$ and hence all equivalent systems are in rational form, thus making the expansion (2) possible.

In particular, we will be using the transformation constructed by stepwise application of a $\mathbb{P}$-*balance between* $x = x_1$ *and* $x = x_2$, defined as:

$$\mathcal{B}(\mathbb{P}(\epsilon), x_1, x_2; x) = \mathbb{I} - \mathbb{P}(\epsilon) + c \frac{x - x_1}{x - x_2} \mathbb{P}(\epsilon), \tag{6}$$

$$c \equiv \begin{cases} 1/x_1 & \text{if } x_1 = \infty, \\ x_2 & \text{if } x_2 = \infty, \\ 1 & \text{otherwise,} \end{cases}$$

where $\mathbb{P}(\epsilon)$ is a projector matrix (that is, $\mathbb{P}^2 = \mathbb{P}$).

Classification of singularities. Following [Mos59], for a system (1) and it's Laurent expansion (2) we define a rational number

$$m_k(\mathbb{A}) = p_k + \frac{rank(\mathbb{A}_{k0})}{n} \tag{7}$$

as the *Moser order* of $\mathbb{A}(x, \epsilon)$ at point $x = x_k$.

Equivalent systems do not necessarily have identical Moser orders. In fact [Mos59] introduces an algorithm that constructs a transformation decreasing $rank(\mathbb{A}_{k0})$ by at least one (thus reducing $m_k(\mathbb{A})$),[2] or certifies that no further order reduction can be achieved.

[2]Note, that if $rank(\mathbb{A}_{k0})$ reaches zero, this means that $\mathbb{A}_{k0}$ is now zero itself, and thus the Poincaré rank p_k was decreased by at least one.

Let us then denote the *minimal order* of $\mathbb{A}(x,\epsilon)$ at $x = x_k$ as:

$$\mu_k(\mathbb{A}) = \min m_k(\mathbb{A}'), \text{ for } \forall\ \mathbb{T}. \tag{8}$$

If $\mu_k(\mathbb{A}) < m_k(\mathbb{A})$ we say that the matrix $\mathbb{A}(x,\epsilon)$ is *Moser-reducible* at $x = x_k$.

With this in mind we can classify a point $x = x_k$ of $\mathbb{A}(x,\epsilon)$ as:

- *regular point*, if $m_k(\mathbb{A}) = 0$;
- *apparent singularity*, if $m_k(\mathbb{A}) > 0$ and $\mu_k(\mathbb{A}) = 0$;
- *regular singularity*, if $0 < \mu_k(\mathbb{A}) \leq 1$;
- *irregular singularity*, if $\mu_k(\mathbb{A}) > 1$.

The matrix $\mathbb{A}(x,\epsilon)$ is called *Fuchsian* if it does not contain irregular singularities at any value of x including ∞.

2.2 Reduction to epsilon form

In the previous section we have introduced the notation and key definitions related to the Fuchsian theory of ODEs. Now we are ready to review the reduction method proposed by Lee in [Lee15]. With its help we can construct a rational transformation $\mathbb{T}(x,\epsilon)$ which converts a system of ordinary differential equations with rational coefficients given by the matrix $\mathbb{A}(x,\epsilon)$ to an equivalent system given by the matrix $\mathbb{M}(x,\epsilon)$ which is Fuchsian and has an *epsilon form* (also called *canonical* in [Hen13]), i.e., $\mathbb{M}(x,\epsilon) = \epsilon\,\mathbb{S}(x)$. When the epsilon form $\mathbb{M}(x,\epsilon)$ of the initial system $\mathbb{A}(x,\epsilon)$ is found we can easily solve it as a Laurent series in ϵ and restore solutions for the initial system $\mathbb{A}(x,\epsilon)$ — which is our ultimate goal — by solving a linear system of equations.

The whole method is performed in the following three steps:

1. Given a matrix $\mathbb{A}(x,\epsilon)$, find an equivalent system $\mathbb{A}'(x,\epsilon)$ and a corresponding transformation $\mathbb{T}(x,\epsilon)$, such that $\mathbb{A}'(x,\epsilon)$ is Fuchsian. We call this step *fuchsification*.
2. Given a Fuchsian matrix $\mathbb{A}(x,\epsilon)$ with eigenvalues of all its residues of the form $n + m\,\epsilon$, where n is integer, find an equivalent system (along with the transformation) which is still Fuchsian, but with residue eigenvalues being all of the form $k\,\epsilon$. We call this step *normalization*.
3. Given a normalized matrix $\mathbb{A}(x,\epsilon)$, find an equivalent matrix $\mathbb{A}'(x)$ in epsilon form, i.e. such that $\mathbb{A}'(x,\epsilon) = \epsilon\,\mathbb{S}'(x)$. We call this step *factorization*.

2.2.1 Fuchsification

To *fuchsify* a system (1) means to find an equivalent Fuchsian system. This, of course, is only possible if $\mathbb{A}(x,\epsilon)$ has no irregular singularities, or in other words $\mu_k(\mathbb{A}) \leq 1$ for all k.[3]

In the case of a single ODE of order n, the minimal Moser order can be computed explicitly from power counting analysis of its coefficients (see the generalization of Fuchs' theorem in [Mos59]). This is not possible for ODE systems like (1). Instead, we have a criterion for Moser-reducibility of the form:

Theorem 1. *If $m_k(\mathbb{A}) > 1$ then the system (1) is Moser-reducible at $x = x_k$ if and only if the polynomial*

$$\Delta^{r_k} \det\left(\frac{\mathbb{A}_{k0}}{\Delta} + \mathbb{A}_{k1} - \lambda\,\mathbb{I}\right), \tag{9}$$

vanishes identically in λ at $x = x_k$, where $r_k = \operatorname{rank}(\mathbb{A}_{k0})$ and $\Delta = x - x_k$ if $x_k \neq \infty$, or $\Delta = 1/x$ if $x_k = \infty$.

When this condition fails the singularity in $x = x_k$ is *irregular*.

In addition to this criterion we have a method for constructing a transformation that lowers the Moser order at x_i, provided that the system is Moser-reducible at that point (possibly at the expense of increasing the Poincaré rank p_j at another point $x = x_j$ by one). This is done by selecting a projector matrix $\mathbb{P}$ equal to a sum of products of a particular subset of (generalized) eigenvectors of $\mathbb{A}_{i0}$ and $\mathbb{A}_{j0}$, and constructing a $\mathbb{P}$-balance between x_i and x_j.

The reader can find the details of this construction in [Lee15], but it is important to note that even if the system is Moser-reducible at x_i, it is only sometimes possible to construct a transformation that lowers $m_i(\mathbb{A})$ without increasing Poincaré rank at x_j. Sometimes the best we can do is to choose x_j to be some regular point (with $p_j = -1$), and use a transformation that decreases $m_i(\mathbb{A})$ at the expense at increasing p_j to 0, effectively introducing an apparent singularity where there was none before.[4]

With this in mind, to reduce a system to Fuchsian form we need to combine the reducibility check with the Moser rank-lowering transformation in stepwise fashion, as follows:

1. Select some point x_k with $m_k(\mathbb{A}) > 1$. If none exist, reduction is complete.
2. Check if $\mathbb{A}(x,\epsilon)$ is reducible at $x = x_k$. If not, fail.
3. Find a transformation $\mathbb{T}(x,\epsilon) = \mathcal{B}(\mathbb{P}(\epsilon), x_k, x_j; x)$ that lowers $m_k(\mathbb{A})$.
4. Apply $\mathbb{T}(x,\epsilon)$ and repeat from Step 1.

[3] We expect this to be often the case in practice, in particular for ODEs corresponding to Feynman integrals which are known to have only logarithmic singularities, hence be solutions of some Fuchsian ODEs.

[4] In practice these apparent singularities are not a major problem, since they are subsequently removed during the normalization step. Still, we try not to introduce them if possible in order to decrease the intermediate expression sizes and to increase the overall performance.

In Fuchsia, this process is implemented by function `fuchsify`; see Section 3.3 for it's usage.

Finally, let us mention that a similar problem of reducing Poincaré ranks and Moser orders of rational matrices was actively studied by Barkatou and co-authors, e.g., see [BP99]. They developed algorithms implementing their method [BP99] which is available in the standard `Maple` package `DEtools` as `moser_reduce` and `super_reduce` routines.

2.2.2 Normalization

To *normalize* a Fuchsian system (1) with matrix residue eigenvalues of the form $n + m\,\epsilon$ (where n is integer), implies to find an equivalent Fuchsian system with residue eigenvalues of the form $m\,\epsilon$.

Just like in the previous step, the normalizing transformations are found by stepwise application of balance transformations (6). We refer the reader to [Lee15, p. 11] for the description of how such balances are constructed, but we will note that this transformation is possible due to these two facts:

- Given a properly selected projector matrix $\mathbb{P}(\epsilon)$, the balance $\mathcal{B}(\mathbb{P}(\epsilon), x_i, x_j; x)$ shifts one of the eigenvalues of $\mathbb{A}_{i0}$ by ± 1, shifts one of the eigenvalues of $\mathbb{A}_{j0}$ by ∓ 1, and does not change the Poincaré ranks at any point.
- Since the system is Fuchsian, the sum of all its matrix residues is zero, and thus, the sum of all residue eigenvalues is zero as well.

Combining these two facts, we perform the normalization by shifting the residue eigenvalues by 1 at each step, until a state is reached where the integer parts of all the eigenvalues are zero.

In Fuchsia, the normalization step is implemented by the function `normalize`. See Section 3.3 for its usage.

It may happen that after fuchsification we obtain a system for which the residue eigenvalues do not fit to the $n+m\,\epsilon$ form neatly. In this case it is sometimes possible to rectify the problem by using some non-linear change of variables. Unfortunately we do not have an automated solution for such cases, and users are expected to find transformations appropriate for their system manually.

2.2.3 Factorization

After the normalization we have obtained a matrix $\mathbb{A}(x, \epsilon)$ with all residue eigenvalues of the form $m\,\epsilon$. The final step is to *factorize* it by finding an equivalent matrix which is by itself proportional to ϵ, so $\mathbb{A}'(x, \epsilon) = \epsilon\,\mathbb{S}(x)$.

A transformation which is constant in x is sufficient for this task. Let $\mathbb{T}(\epsilon)$ be such a transformation, then according to (5), we have:

$$\mathbb{A}'(x,\epsilon) \equiv \epsilon \sum_i \frac{\mathbb{S}_i}{x - x_i} = \sum_i \mathbb{T}^{-1}(\epsilon) \frac{\mathbb{A}_{i0}(\epsilon)}{x - x_i} \mathbb{T}(\epsilon), \tag{10}$$

or explicitly

$$\mathbb{S}_i = \mathbb{T}^{-1}(\epsilon)\frac{\mathbb{A}_{i0}(\epsilon)}{\epsilon}\mathbb{T}(\epsilon). \tag{11}$$

Since $\mathbb{S}_i$ in (11) is the same no matter what values ϵ takes, we can say that:

$$\mathbb{S}_i = \mathbb{T}^{-1}(\epsilon)\frac{\mathbb{A}_{i0}(\epsilon)}{\epsilon}\mathbb{T}(\epsilon) = \mathbb{T}^{-1}(\mu)\frac{\mathbb{A}_{i0}(\mu)}{\epsilon}\mathbb{T}(\mu), \tag{12}$$

from where we obtain a system of linear equations for $\mathbb{T}(\epsilon,\mu) \equiv \mathbb{T}(\epsilon)\,\mathbb{T}^{-1}(\mu)$:

$$\frac{\mathbb{A}_{i0}(\epsilon)}{\epsilon}\mathbb{T}(\epsilon,\mu) = \mathbb{T}(\epsilon,\mu)\frac{\mathbb{A}_{i0}(\mu)}{\mu}, \text{for all } i. \tag{13}$$

Solving this system for $\mathbb{T}(\epsilon,\mu)$ we can reconstruct the initial transformation as $\mathbb{T}(\epsilon) = \mathbb{T}(\epsilon,\mu_0)$, where μ_0 can be chosen arbitrary as long as $\mathbb{T}(\epsilon)$ will come out invertible.

In general, the solution for $\mathbb{T}(\epsilon,\mu)$ can have multiple free variables aside from just μ. We choose to set all of them to random small integers, preferably zeros, which keeps the resulting matrix $\mathbb{S}(\epsilon)$ simple.

In `Fuchsia` the factorization step is implemented as the `factorize` routine (again, see Section 3.3 for its usage).

2.2.4 Block-triangular form

It often happens that a matrix which defines an ODEs is sparse, i.e. it has many zeros, and can be shuffled into block-triangular form with small blocks. It is possible to exploit this fact to considerably speed up fuchsification and normalization.

Let us consider a block-triangular matrix of this form:

$$\mathbb{A}(x,\epsilon) = \begin{pmatrix} \mathbb{A}^{(11)} & 0 & 0 & 0 \\ \mathbb{A}^{(21)} & \mathbb{A}^{(22)} & 0 & 0 \\ \vdots & \cdots & \ddots & 0 \\ \mathbb{A}^{(m1)} & \mathbb{A}^{(m2)} & \cdots & \mathbb{A}^{(mm)} \end{pmatrix}, \tag{14}$$

where $\mathbb{A}^{(ab)}(x,\epsilon)$ is a sub-matrix of size $n_a \times n_b$.

We start by reducing **diagonal blocks** of this matrix, $\mathbb{A}^{(ab)}(x,\epsilon)$, to epsilon form. This can be done by treating each diagonal block as an independent matrix, and proceeding as described in the previous sections. Since the characteristic polynomial of a block-triangular matrix is a product of characteristic polynomials of its diagonal blocks, i.e.,

$$\det(\mathbb{A}(x,\epsilon) - \lambda\mathbb{I}) = \det(\mathbb{A}^{(11)}(x,\epsilon) - \lambda\mathbb{I}) \cdot \ldots \cdot \det(\mathbb{A}^{(mm)}(x,\epsilon) - \lambda\mathbb{I}), \tag{15}$$

once we have normalized the diagonal blocks, the whole matrix becomes normalized as well (but not necessarily Fuchsian yet).

Next, we fuchsify **off-diagonal blocks** given by rectangular matrices $\mathbb{A}^{(ab)}(x,\epsilon)$, $a > b$. To do this, let us look at the parts of (1) related to such a block:

$$\begin{cases} \partial_x \mathbf{J}^{(a)} = \mathbb{A}^{(aa)}\,\mathbf{J}^{(a)} + \ldots \\ \partial_x \mathbf{J}^{(b)} = \mathbb{A}^{(b\,a)}\,\mathbf{J}^{(a)} + \mathbb{A}^{(bb)}\,\mathbf{J}^{(b)} + \ldots \end{cases} \tag{16}$$

If $\mathbb{A}^{(b\,a)}(x,\epsilon)$ has a singularity of Poincaré rank $r > 0$ at $x = x_k$, then to reduce r we can apply this basis transformation:

$$\mathbf{J}^{(b)} = \mathbf{J}'^{(b)} + \Delta^{-r}\mathbb{D}\,\mathbf{J}^{(a)}, \tag{17}$$

where $\mathbb{D}$ is some constant matrix, and $\Delta = x - x_k$ if $x_k \neq \infty$, or $\Delta = 1/x$ if $x_k = \infty$. This transformation changes $\mathbb{A}_{k0}^{(b\,a)}(\epsilon)$ into

$$\mathbb{A}_{k0}'^{(ba)}(\epsilon) = \mathbb{A}_{k0}^{(ba)}(\epsilon) + r\,\mathbb{D} + \mathbb{A}_{k0}^{(bb)}(\epsilon)\,\mathbb{D} - \mathbb{D}\,\mathbb{A}_{k0}^{(aa)}(\epsilon) \tag{18}$$

If both $\mathbb{A}_{k0}^{(aa)}(\epsilon)$ and $\mathbb{A}_{k0}^{(bb)}(\epsilon)$ have been factorized then it is always possible to solve the right-hand side of this equation for $\mathbb{D}$, thus reducing the Poincaré rank of $\mathbb{A}'^{(ba)}$ by one. Moreover, this transformation only affects $\mathbb{A}^{(bi)}$ for $i \leq a$ and $\mathbb{A}^{(ia)}$ for $i > b$, therefore if we will sequentially apply the transformation (17) for each off-diagonal block, starting from the first row to the last, and from the last column to the first, we will obtain a fully Fuchsian (and still normalized) matrix.

Finally, we factorize the whole matrix as described in Section 2.2.3, thus completing the transformation to epsilon form.

In `Fuchsia` the process of shuffling a matrix to its shortest lower block-diagonal form is performed by the `block_triangular_form` routine; fuchsification and normalization of diagonal blocks is done by the `reduce_diagonal_blocks`; fuchsification of the remaining off-diagonal blocks is done by the `fuchsify_off_diagonal_blocks`.

3 Using Fuchsia

3.1 Installation

To run `Fuchsia` one needs `SageMath` [Dev16] version 7.0 or higher to be installed on the computer. This task can be accomplished by following installation instructions available at the website `http://www.sagemath.org`. [5]

`SageMath` is a free and open-source Computer Algebra System licensed under GPL. It is written in `Python` 2.7 and combines together a number of existing open-source mathematical systems and libraries like `Maxima`, `Singular`, and others with the goal of providing the best free CAS. In particular our code heavily relies on the interface to `Maxima` [Max14].

We also should obtain a file `fuchsia.py` which can be downloaded from the website `https://github.com/gituliar/fuchsia`, where many examples of usage and reduced matrices are also collected.

[5] Some `Linux` distributions have `SageMath` available in their package repositories; we do not recommend using those. A number of `Maxima` releases contain bugs which `Fuchsia` is sensitive to, and so far the official `SageMath` builds have avoided those releases (unlike some `Linux` distributions).

3.2 Usage from the command line

To run Fuchsia use the command[6]

```
$ sage -python fuchsia.py <action> <options>
```

where

- `<action>` is one of the algorithms described in the previous section, i.e., `fuchsify`, `normalize`, `factorize`, or auxiliary action `transform`, which applies a user-defined transformation to the given matrix.
- `<options>` are action-dependent options described in the help message printed with the help of `fuchsia -h` command.

In the following we provide a complete help information printed by `fuchsia -h`:

```
Fuchsia v16.11.14
Authors: Oleksandr Gituliar, Vitaly Magerya

Usage:
    fuchsia [-hv] [--use-maple] [-f <fmt>] [-l <path>] [-P <path>]
            <command> <args>...

Commands:
    reduce [-x <name>] [-e <name>] [-m <path>] [-t <path>] <matrix>
        find an epsilon form of the given matrix

    fuchsify [-x <name>] [-m <path>] [-t <path>] <matrix>
        find a transformation that will transform a given matrix
        into Fuchsian form

    normalize [-x <name>] [-e <name>] [-m <path>] [-t <path>] <matrix>
        find a transformation that will transform a given Fuchsian
        matrix into normalized form

    factorize [-x <name>] [-e <name>] [-m <path>] [-t <path>] <matrix>
        find a transformation that will make a given normalized
        matrix proportional to the infinitesimal parameter

    sort [-m <path>] [-t <path>] <matrix>
        find a block-triangular form of the given matrix

    transform [-x <name>] [-m <path>] <matrix> <transform>
        transform a given matrix using a given transformation

    changevar [-x <name>] [-m <path>] <matrix> <expr>
        transform matrix by susbtituting free variable by a
```

[6] For brevity, we use a shortcut `fuchsia` which is equivalent to `sage -python fuchsia.py`.

```
        given expression

Options:
    -h          show this help message
    -f <fmt>    matrix file format: mtx or m (default: mtx)
    -l <path>   write log to this file
    -v          produce a more verbose log
    -P <path>   save profile report into this file
    -x <name>   use this name for the free variable (default: x)
    -e <name>   use this name for the infinitesimal parameter (default: eps)
    -m <path>   save the resulting matrix into this file
    -t <path>   save the resulting transformation into this file
    --use-maple speed up calculations by using Maple when possible

Arguments:
    <matrix>    read the input matrix from this file
    <transform> read the transformation matrix from this file
    <expr>      arbitrary expression
```

Simple and more advanced results obtained with the help of Fuchsia are located in the directory `examples`. There you will find many examples of original matrices together with generated transformations which lead to the epsilon form of corresponding matrices. Another example of applying Fuchsia to find master integrals for next-to-leading order contributions to splitting functions in QCD were discussed in [GM16].

3.3 Usage from SageMath or Python

You can also use Fuchsia as a library by starting the `SageMath` prompt and importing the `fuchsia` module like this:

```
$ sage

 SageMath Version 7.1, Release Date: 2016-03-20
 Type "notebook()" for the browser-based notebook interface.
 Type "help()" for help.

sage: from fuchsia import *
```

In order to give an example for the API, let us try to reduce a simple matrix. For the list of functions available after import, please, read the next section.

```
sage: x, eps = var("x eps")
sage: M = matrix([
....:   [(2-eps)/x, 0, 0],
....:   [x/(x-1), eps/x, 0],
....:   [(1+2*eps)/x**3, 0, (1+eps)/x/(x+1)]
....: ])
```

First, let us see where the singularities of this matrix are located:

```
sage: singularities(M, x)
{-1: 0, 0: 2, 1: 0, +Infinity: 1}
```

So, 4 singularities in total, with the Poincaré rank being 2 at $\mathtt{x} = 0$, 1 at $\mathtt{x} = \infty$ and 0 everywhere else. To get rid of non-zero ranks (thus transforming the system into Fuchsian form) we will need to *fuchsify* this matrix as:

```
sage: Mf, Tf = fuchsify(M, x)
sage: Mf
[ -(eps - 2)/x           0                          0]
[   -1/(x - 1)  (eps -1)/x                          0]
[(2*eps + 1)/x           0  (eps + 2*x + 3)/(x^2 + x)]
sage: singularities(Mf, x)
{-1: 0, 0: 0, 1: 0, +Infinity: 0}
```

Now, let us take a look at the eigenvalues of `Mf` residues:

```
sage: [matrix_residue(Mf, x, x0).eigenvalues()
....:   for x0 in [-1, 0, 1, Infinity]]
[[-eps - 1, 0, 0],
 [-eps + 2, eps - 1, eps + 3],
 [0, 0, 0],
 [-eps + 1, eps - 2, -2]]
```

Many of these eigenvalues are not equal to zero in the limit $\mathtt{eps} \to 0$, so `Mf` is not normalized. It is, however, the case that all of the eigenvalues are of the form $n + m * \mathtt{eps}$, so there is a chance that we will be able to normalize `Mf`. Let us try:

```
sage: Mn, Tn = normalize(Mf, x, eps)
sage: Mn
[-eps/x                                                                  ...
[(4*eps^3 - 8*eps^2 - (4*eps^2 - 6*eps + 3)*x + 5*eps)/((4*eps^3 - ...
[((2*eps + 1)*x + 3*eps + 1)/(x^2 + x)                                   ...
sage: [matrix_residue(Mn, x, x0).eigenvalues()
....:   for x0 in [-1, 0, 1, Infinity]]
[[-eps, 0, 0],
 [-eps, eps, eps],
 [0, 0, 0],
 [-eps, eps, 0]]
```

So, the matrix is normalized, but it grew quite a bit larger. This happens. Sometimes it is possible to simplify it a bit:

```
sage: Ms, Ts = simplify_by_jordanification(Mn, x)
sage: Ms
[               -eps/x      0              0]
[            1/(x - 1)  eps/x              0]
[1/2*(eps + 1)/(x + 1)      0  eps/(x^2 + x)]
```

That is much better.

Finally, we need to *factorize* `Ms` to complete the reduction:

```
sage: Mr, Tr = factorize(Ms, x, eps)
sage: Mr
[         -eps/x      0              0]
[1/4*eps/(x - 1)  eps/x              0]
[5/8*eps/(x + 1)      0  eps/(x^2 + x)]
```

This is the fully transformed matrix. As you can see, it is both proportional to `eps` and Fuchsian. To make sure we got everything right, we can double-check the full transformation:

```
sage: T = (Tf*Tn*Ts*Tr).simplify_rational()
sage: (Mr - transform(M, x, T)).is_zero()
True
```

Note that we have used the construct `(A - B).is_zero()` to compare matrices instead of the more obvious `bool(A == B)`. This is a `SageMath` idiosyncrasy; the more obvious way compares symbolic matrices only structurally.

Of course, you do not need to walk through all these steps yourself every time. Normally, you just need to call this one function to do all the reduction work:

```
sage: MM, TT = epsilon_form(M, x, eps)
sage: MM
[                           -eps/x      0              0]
[                 1/4*eps/(x - 1)  eps/x              0]
[1/4*(9*eps*x + 13*eps)/(x^2 + x)      0  eps/(x^2 + x)]
sage: (MM - transform(M, x, TT)).is_zero()
True
```

Notice that this matrix is slightly more complex than the one we have obtained step by step above. This also happens. The final form we are computing is not unique, and it will be different depending on the precise sequence of reduction steps you have taken.

Additionally, many of the transformations take a special *seed* parameter to control the order of operations they perform internally. By supplying different seeds, you will obtain different results as well.

3.3.1 Function reference

`epsilon_form`($\mathbb{M}, x, epsilon, seed = 0$) ***(function)***

Fully reduces a system of equations defined by a matrix $\mathbb{M}$, an independent variable x, and an infinitesimal parameter *epsilon*. Returns a pair of values: the transformed matrix $\mathbb{M}'$ and the transformation matrix $\mathbb{T}$. Raises *FuchsiaError*, if the system is irreducible.

The reduction is performed by first converting $\mathbb{M}$ to block-triangular form, then reducing the diagonal blocks via `fuchsify`, `normalize` and `factorize`, reducing off-diagonal blocks as described in Section 2.2.4, and finally factorizing *epsilon* via `factorize`.

`fuchsify`($\mathbb{M}, x, seed = 0$) ***(function)***

Reduces a system defined by a matrix $\mathbb{M}$ and an independent variable x to Fuchsian form. That is, it makes sure that the Poincaré ranks of all singularities of the transformed matrix $\mathbb{M}'$ are 0. Returns a pair of values: the transformed matrix $\mathbb{M}'$ and the transformation $\mathbb{T}$. If the system is irreducible, it raises *FuchsiaError*.

`normalize`($\mathbb{M}, x, epsilon, seed = 0$) ***(function)***

Transforms a Fuchsian system defined by a matrix $\mathbb{M}$, an independent variable x and, an infinitesimal parameter *epsilon* to a normalized form. That is, it makes sure that real parts of the eigenvalues of all matrix residues of the transformed matrix $\mathbb{M}'$ lie in the range $[-1/2, 1/2)$ in the limit $epsilon \to 0$. Returns a pair of values: the transformed matrix $\mathbb{M}'$ and the transformation $\mathbb{T}$. If such a transformation can not be found, it raises *FuchsiaError*.

`factorize`($\mathbb{M}, x, epsilon, seed = 0$) ***(function)***

Transforms a normalized system defined by a matrix $\mathbb{M}$, an independent variable x, and an infinitesimal parameter *epsilon* so that the transformed matrix $\mathbb{M}'$ is proportional to *epsilon*. Returns a pair of values: the transformed matrix $\mathbb{M}'$ and the transformation $\mathbb{T}$. If such a transformation can not be found, it raises *FuchsiaError*.

`block_triangular_form`($\mathbb{M}$) ***(function)***

Transforms a matrix $\mathbb{M}$ into a lower block-triangular form.

Returns three values: a transformed matrix $\mathbb{M}'$, a transformation matrix $\mathbb{T}$, and a list of tuples (m_i, n_i), where n_i is the size of i-th diagonal block, and $m_i = \sum_{j=1}^{i-1} n_j$. The tuple list represents block structure of $\mathbb{M}'$; it is used by the next two functions, where it is passed as the argument B.

`reduce_diagonal_blocks`($\mathbb{M}, x, epsilon, B = None, seed = 0$) ***(function)***

Finds a transformation that reduces diagonal blocks of $\mathbb{M}$ into epsilon form. If B is not provided, it transforms $\mathbb{M}$ into lower triangual form before reduction. Otherwise it assumes $\mathbb{M}$ blocks are described by B.

Returns two values: a transformed matrix $\mathbb{M}$ and a transformation matrix $\mathbb{T}$.

`fuchsify_off_diagonal_blocks`($\mathbb{M}, x, epsilon, r = None$) ***(function)***

Given a matrix M with diagonal blocks in epsilon form, it transforms off-diagonal

blocks in to Fuchsian form. If B is not provided, it transforms $\mathbb{M}$ into lower triangual form before reduction. Otherwise it assumes that blocks of $\mathbb{M}$ are described by B.

Returns two values: a transformed matrix $\mathbb{M}$ and a transformation matrix $\mathbb{T}$.

`simplify_by_jordanification`$(\mathbb{M}, x)$ ***(function)***

Tries to simplify a system defined by a matrix $\mathbb{M}$ and an independent variable x by constant transformations that transform leading expansion coefficients of $\mathbb{M}$ into their Jordan forms. Returns a pair of values: the simplified matrix $\mathbb{M}'$ and the transformation $\mathbb{T}$. If none of the attempted transformations reduces the complexity of $\mathbb{M}$ (as measured by `matrix_complexity`), it returns the original matrix and the identity transformation.

`simplify_by_factorization`$(\mathbb{M}, x)$ ***(function)***

Tries to simplify a system defined by a matrix $\mathbb{M}$ and an independent variable x by a constant transformation that extracts common factors found in $\mathbb{M}$ (if any). Returns a pair of values: the simplified matrix $\mathbb{M}'$ and the transformation $\mathbb{T}$.

`matrix_complexity`$(\mathbb{M})$ ***(function)***

This function is used as a measure of matrix complexity by `fuchsify` and simplification functions. Currently it is defined as the length of textual representation of matrix $\mathbb{M}$.

`balance`$(\mathbb{P}, x_1, x_2, x)$ ***(function)***

Returns a *balance* transformation between points $x = x_1$ and $x = x_2$ using the projector matrix $\mathbb{P}$. See eq. (6) for the definition of a *balance*.

`transform`$(\mathbb{M}, x, \mathbb{T})$ ***(function)***

Transforms a system defined by a matrix $\mathbb{M}$ and an independent variable x using the transformation matrix $\mathbb{T}$ as specified by eq. (5). Returns the transformed matrix $\mathbb{M}'$.

`balance_transform`$(\mathbb{M}, \mathbb{P}, x_1, x_2, x)$ ***(function)***

Same as `transform`$(\mathbb{M}, x,$ `balance`$(\mathbb{P}, x_1, x_2, x))$, but implemented more efficiently: since the inverse of `balance`$(\mathbb{P}, x_1, x_2, x)$ is `balance`$(\mathbb{P}, x_2, x_1, x)$, this function can avoid a time-consuming matrix inversion operation that `transform` must perform.

`singularities`$(\mathbb{M}, x)$ ***(function)***

Finds values of x around which the matrix $\mathbb{M}$ has a singularity in x. Returns a dictionary with $\{x_i : p_i\}$ entries, where p_i is the Poincaré rank of $\mathbb{M}$ at $x = x_i$. The set of singular points can include `Infinity`, if $\mathbb{M}$ has a singularity at $x \to \infty$.

`matrix_c0`$(\mathbb{M}, x, x_0, p)$ ***(function)***

Returns the 0-th coefficient of the series expansion of a matrix $\mathbb{M}$ around $x = x_0$, assuming the Poincaré rank of $\mathbb{M}$ at that point is p. If x_0 is `Infinity`, it returns minus the coefficient at the highest power of x. In other words, it returns $\mathbb{A}_{k0}$ from eq. (2).

`matrix_c1`($\mathbb{M}, x, x_0, p$) ***(function)***

Returns the 1-th coefficient of the series expansion of a matrix $\mathbb{M}$ around $x = x_0$, assuming the Poincaré rank of $\mathbb{M}$ at that point is p. If x_0 is `Infinity`, it returns minus the coefficient at the second-to-highest power of x. In other words, it returns $\mathbb{A}_{k1}$ from eq. (2).

`matrix_residue`($\mathbb{M}, x, x_0$) ***(function)***

Returns a residue of a matrix $\mathbb{M}$ at $x = x_0$, assuming that the Poincaré rank of $\mathbb{M}$ at $x = x_0$ is 0. Returns matrix residue at infinity if x_0 is `Infinity`.

This is the same as `matrix_c0`($\mathbb{M}, x, x_0, 0$).

`export_matrix_to_file`($filename, \mathbb{M}, fmt =$ `"mtx"`) ***(function)***

Writes a matrix $\mathbb{M}$ to a file $filename$ using MatrixMarket array format if fmt is `"mtx"` (which is the default), or Mathematica format if fmt is `"m"`.

`import_matrix_from_file`($filename$) ***(function)***

Reads a symbolic matrix from a file $filename$. Both Mathematica and MatrixMarket array formats are supported. The exact format will be autodetected.

`setup_fuchsia`($verbosity = 0, use\_maple =$ `False`) ***(function)***

Modifies some of the `Fuchsia` inner workings. In particular, it sets $verbosity$ to 2 to enable verbose logging, 1 to enable normal logging, and 0 to only log errors.

Set use_maple to `True` to enable usage of `Maple` to speed up calculations when possible; this may be particularly beneficial for big matrices, and for matrices with singularities at complex points.

FuchsiaError ***(class)***

This is the class of exceptions raised by the `Fuchsia` routines. It indicates the inability to perform the requested reduction.

4 Summary

In this paper we have presented `Fuchsia`, a program for reducing differential equations for Feynman master integrals to the epsilon form based on the Lee algorithm [Lee15] which consists of three main computational steps: fuchsification, normalization, and factorization. `Fuchsia` is open-source nature and depends on free software tools only: the programming language `Python` and the computer algebra system `Maxima/Sage`, which makes it available for everyone. Unfortunately, for some heavy-duty computations one needs to switch from `Maxima` to `Maple`, which is more powerful for some tasks, hence the access to latter is desirable. Though it helps a lot in some situation, `Maple` is not a universal solution and is not able to process huge intermediate expressions which arise during the reduction process of some advanced examples. It happens due to inability to work with polynomials which contain algebraic numbers in their coefficients (even in the case of complex coefficients the performance drastically decreases).

Despite of the discussed limitation `Fuchsia` shows a great performance in many cases. It is possible due to the optimization for block-triangular (or sparse) matrices, which allows

to reduce relatively large matrices: the reduction of 74×74 matrix with 20 rational and complex singular points and at most 3×3 coupled blocks takes about an hour on a laptop with Intel i5 CPU.

Acknowledgment

We are gratefully thankful for advanced examples of differential equations provided by Roman Lee and Costas Papadopolous. We also appreciate useful suggestions from Sven Moch during writing of this paper.

This work has been supported by the Deutsche Forschungsgemeinschaft in Sonderforschungsbereich 676 *Particles, Strings, and the Early Universe* and by the Narodowe Centrum Nauki with the Sonata Bis grant DEC-2013/10/E/ST2/00656.

References

[ABB+16] J. Ablinger, A. Behring, J. Blümlein, A. De Freitas, A. von Manteuffel, and C. Schneider. Calculating Three Loop Ladder and V-Topologies for Massive Operator Matrix Elements by Computer Algebra. *Comput. Phys. Commun.*, 202:33–112, 2016. arXiv:1509.08324.

[BP99] M. Barkatou and E. Pflügel. An algorithm computing the regular formal solutions of a system of linear differential equations. *Journal of Symbolic Computation*, 28(45):569 – 587, 1999. doi:http://dx.doi.org/10.1006/jsco.1999.0315.

[CT81] K. Chetyrkin and F. Tkachov. Integration by Parts: The Algorithm to Calculate beta Functions in 4 Loops. *Nucl. Phys.*, B192:159–204, 1981. doi:10.1016/0550-3213(81)90199-1.

[Dev16] The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 7.0)*, 2016. http://www.sagemath.org.

[GLZ16] A. Georgoudis, K. J. Larsen, and Y. Zhang. Azurite: An algebraic geometry based package for finding bases of loop integrals. 2016. arXiv:1612.04252.

[GM16] O. Gituliar and V. Magerya. Fuchsia and master integrals for splitting functions from differential equations in QCD. *PoS*, LL2016:030, 2016. arXiv:1607.00759.

[Hen13] J. Henn. Multiloop integrals in dimensional regularization made simple. *Phys. Rev. Lett.*, 110:251601, 2013. arXiv:1304.1806.

[Hen15] J. Henn. Lectures on differential equations for Feynman integrals. *J. Phys.*, A48:153001, 2015. arXiv:1412.2296.

[Kot91a] A. Kotikov. Differential equation method: The Calculation of N point Feynman diagrams. *Phys. Lett.*, B267:123–127, 1991. doi:10.1016/0370-2693(91)90536-Y.

[Kot91b] A. Kotikov. Differential equations method: New technique for massive Feynman diagrams calculation. *Phys. Lett.*, B254:158–164, 1991. doi:10.1016/0370-2693(91)90413-K.

[Kot91c] A. Kotikov. Differential equations method: The Calculation of vertex type Feynman diagrams. *Phys. Lett.*, B259:314–322, 1991. doi:10.1016/0370-2693(91)90834-D.

[Lap00] S. Laporta. High precision calculation of multiloop Feynman integrals by difference equations. *Int. J. Mod. Phys.*, A15:5087–5159, 2000. arXiv:hep-ph/0102033.

[Lee12] R. Lee. Presenting LiteRed: a tool for the Loop InTEgrals REDuction. 2012. arXiv:1212.2685.

[Lee14] R. Lee. LiteRed 1.4: a powerful tool for reduction of multiloop integrals. *J.Phys.Conf.Ser.*, 523:012059, 2014. arXiv:1310.1145.

[Lee15] R. Lee. Reducing differential equations for multiloop master integrals. *JHEP*, 04:108, 2015. arXiv:1411.0911.

[Max14] Maxima. Maxima, a computer algebra system. version 5.35.1, December 2014. URL: http://maxima.sourceforge.net/.

[Mey16a] C. Meyer. Evaluating multi-loop Feynman integrals using differential equations: automatizing the transformation to a canonical basis. *PoS*, LL2016:028, 2016. URL: http://pos.sissa.it/cgi-bin/reader/contribution.cgi?id=260/028.

[Mey16b] C. Meyer. Transforming differential equations of multi-loop Feynman integrals into canonical form. 2016. arXiv:1611.01087.

[Mos59] J. Moser. The order of a singularity in fuchs' theory. *Mathematische Zeitschrift*, 72(1):379–398, 1959. doi:10.1007/BF01162962.

[Pap14] C. Papadopoulos. Simplified differential equations approach for Master Integrals. *JHEP*, 07:088, 2014. arXiv:1401.6057.

[Pra17] M. Prausa. epsilon: A tool to find a canonical basis of master integrals. 2017. arXiv:1701.00725.

[Smi06] V. Smirnov. *Feynman Integral Calculus*. Springer, 2006. doi:10.1007/3-540-30611-0.

[Smi08] A. Smirnov. Algorithm FIRE – Feynman Integral REduction. *JHEP*, 10:107, 2008. arXiv:0807.3243.

[Smi15] A. Smirnov. FIRE5: a C++ implementation of Feynman Integral REduction. *Comput. Phys. Commun.*, 189:182–191, 2015. arXiv:1408.2372.

[SS13] A. Smirnov and V. Smirnov. FIRE4, LiteRed and accompanying tools to solve integration by parts relations. *Comput. Phys. Commun.*, 184:2820–2827, 2013. arXiv:1302.5885.

[Tan15] L. Tancredi. Integration by parts identities in integer numbers of dimensions. A criterion for decoupling systems of differential equations. *Nucl. Phys.*, B901:282–317, 2015. arXiv:1509.03330.

[vMS12] A. von Manteuffel and C. Studerus. Reduze 2 - Distributed Feynman Integral Reduction. 2012. arXiv:1201.4330.