LIPS: *p*-adic and singular phase space

Giuseppe De Laurentis
Paul Scherrer Institut, CH-5232 Villigen PSI, Switzerland
E-mail: giuseppe.de-laurentis@psi.ch

Abstract. I present new features of the open-source Python package `lips`, which leverages the newly developed `pyadic` and `syngular` libraries. These developments enable the generation and manipulation of massless phase-space configurations beyond real kinematics, defined in terms of four-momenta or Weyl spinors, not only over complex numbers ($\mathbb{C}$), but now also over finite fields ($\mathbb{F}_p$) and p-adic numbers ($\mathbb{Q}_p$). The package also offers tools to evaluate arbitrary spinor-helicity expressions in any of these fields. Furthermore, using the algebraic-geometry submodule, which utilizes `Singular` [1] through the Python interface `syngular`, one can define and manipulate ideals in spinor variables, enabling the identification of irreducible surfaces where scattering amplitudes have well-defined zeros and poles. As an example application, I demonstrate how to infer valid partial-fraction decompositions from numerical evaluations.

1. Introduction

High-multiplicity loop-level amplitude computations involve significant algebraic complexity, which is usually side-stepped via numerical routines. Yet, when available, compact analytical expressions can display improved numerical stability and reduced evaluation times. Moreover, much of the recent progress in the computation of loop-corrections to scattering amplitudes has been achieved thanks to finite-field methods [2,3]. As these numerical computations are unsuited for direct phenomenological applications, analytic expressions must be recovered, so that they can then be evaluated with floating-point numbers.

An important role to manifest the analytic properties of scattering amplitudes is played by the spinor-helicity formalism. A classical example is that numerators in gauge-theoretic amplitudes, such as quantum chromodynamics, mitigate the degree of divergences which would naively be expected from Feynman propagators, namely from factors of $1/s_{ij}$ to $1/\sqrt{s_{ij}}$. Relaxing the constraint of real kinematics, one realizes that these divergences are in fact either purely holomorphic spinor contractions $\langle ij\rangle$ or purely anti-holomorphic ones $[ij]$ (with $s_{ij} = \langle ij\rangle[ji]$).

It has been shown that significant insights into the analytic structure of amplitudes can be obtained from tailored numerical evaluations in singular limits [4,5]. For example, the rational prefactors appearing in the planar two-loop amplitude for the process $0 \to q^+\bar{q}^-\gamma^+\gamma^+\gamma^+$ with a closed fermion loop [6,7] can be simplified to just the following two functions

$$\frac{\langle 23\rangle[23]\langle 24\rangle[34]}{\langle 15\rangle\langle 34\rangle\langle 45\rangle\langle 4|1+5|4]} + (45 \to 54), \tag{1}$$

$$\frac{\langle 13\rangle[13]\langle 24\rangle[45]}{\langle 13\rangle\langle 34\rangle\langle 45\rangle\langle 4|1+3|4]} + (45 \to 54) - \frac{\langle 12\rangle[13]\langle 23\rangle^2}{\langle 13\rangle\langle 24\rangle\langle 25\rangle\langle 34\rangle\langle 35\rangle}, \tag{2}$$

together with those obtained by closing the vector space generated by these two functions under the permutations of the photons (legs 3, 4, and 5). In a soon-to-appear paper, we obtain analogous expressions for the full-color two-loop $0 \to q\bar{q}\gamma\gamma\gamma$ amplitude [8].

2. Lips: a phase-space generator for theory computations

Phase-space generators in high-energy physics traditionally describe the kinematics of physical processes at colliders, meaning they provide real-valued phase-space configurations. Yet, from a theoretical standpoint, the analytic properties of scattering amplitudes in perturbative quantum field theory are better understood in the complex plane. This motivates the development of a phase-space generator that exploits the additional freedom of complex kinematics.

The package `lips` (short for Lorentz invariant phase space) provides a phase-space generator and manipulator that is tailored to the needs of modern theoretical calculations. The package is designed to work for processes of arbitrary multiplicity, although at present it can easily handle massless particles only. Nevertheless, massive particles can already be described in terms of a pair of massless decay products. Use cases include: 1) generation of phase-space points over complex numbers ($\mathbb{C}$), finite fields ($\mathbb{F}_p$), and p-adic numbers ($\mathbb{Q}_p$); 2) on-the-fly evaluation of arbitrary spinor-helicity expressions; 3) construction of special kinematic configurations; 4) algebro-geometric analysis of irreducible varieties in kinematic space.

Live examples powered by `binder` [9] are accessible through the badge on the GitHub page.[1]

2.1. Installation

The required language is Python 3, with version $\geq$ 3.8 being recommended. The package is available through the Python Package Index,[2] and thus can be installed via `pip`.

```
pip install --upgrade lips
```

Alternatively, the source code can be cloned from GitHub, and then installed with `pip`.

```
pip install -e path/to/repository
```

The option `-e` ensures that changes to the source code are reflected without having to reinstall the package, for instance after invoking `git pull` to download an update.

2.2. Dependencies

The Python ecosystem provides a rich variety of third-party, open-source libraries for scientific computing. Among these `lips` depends on `NumPy` [10], whose `ndarray` class is used to describe all Lorentz tensors, `mpmath` [11], from which multi-precision real and complex numbers are imported, and `sympy` [12], for symbolic manipulations. Dependencies are declared in the file `setup.py` and are installed automatically through `pip`. The only exception is `Singular` [1], which is optional, and needs to be installed separately. Additionally two new, open-source dependencies are used, namely `pyadic` and `syngular`. They can be used independently of `lips`.

2.2.1. `pyadic` Finite fields have become a staple of multi-loop computations. More recently, the idea to use p-adic numbers was introduced, in order to rescue a non-trivial absolute value while maintaining integer arithmetic [5]. The package `pyadic` provides classes for finite fields (`ModP`) and p-adic numbers (`PAdic`), instantiated as shown, as well as related algorithms.

```
ModP(number, prime)
PAdic(number, prime, digits, valuation=0)
```

Besides standard arithmetic operations, square roots are also available in the functions `finite_field_sqrt` and `padic_sqrt`. These are not guaranteed to be in the field and may

[1] At the URL github.com/GDeLaurentis/lips

[2] At the URL pypi.org/project/lips/

	"mpc"	"gaussian rational"	"finite field"	"padic"
prime	✗	✗	✓	✓
digits	✓	✗	✗	✓

Table 1. Used (✓) and discarded (✗) arguments for `Field`.

return a `FieldExtension` object. Only field extensions by a single square root are currently implemented. Moreover, the p-adic logarithm is also implemented in the function `padic_log`.

For p-adic numbers, numerical precision is explicitly tracked as an error $\mathcal{O}(p^k)$ term, meaning all displayed digits are, by default, significant digits. This allows one to perform computations in singular configurations while keeping track of the numerical uncertainty. Nevertheless, the parameter `fixed_relative_precision` can be switched to `True` to emulate the usual floating-point behavior, with numerical noise being appended to numbers in case of precision loss.

Rational reconstruction algorithms from $\mathbb{F}_p$ and $\mathbb{Q}_p$ to $\mathbb{Q}$ are also provided with the function `rationalise`, which takes an optional keyword argument `algorithm` to toggle between maximal quotient reconstruction (`MQRR`) and lattice reduction (`LGRR`) [13–15].

It is also worth mentioning that an alternative implementation of finite fields is available in the package `galois` [16]. This is particularly useful when dealing with large `ndarrays`.

2.2.2. `Singular`/`syngular` The submodule `algebraic_geometry` of `lips` requires `Singular` [1]. To facilitate computations, an object-oriented Python interface is provided in the package `syngular` [17]. Like `pyadic`, this can be used independently of `lips`. The main classes currently implemented are `Ideal`, `Ring` and `QuotientRing`. They provide easy access to several functions implemented in `Singular` in a pythonic way. For instance, we have ideal addition (`+`), product (`*`), quotient (`/`), membership (`in`), intersection (`&`), equality (`==`), *etc.* More methods are available with self-explanatory names, e.g. `primary_decomposition`, which calls `primdecGTZ`.

2.3. Basic usage

The main class provided by `lips` is the `Particles` class. It describes the phase space of massless particles with given `multiplicity` in 4 dimensions. By default, the 4-momenta are taken to be complex valued. A specific choice of number field can be passed as a keyword parameter.

```
Particles(multiplicity, field=Field(name, prime, digits))
```

The generated phase-space point will satisfy on-shell relations ($p_i^\mu p_{i,\mu} = 0 \; \forall \; i$) and momentum conservation ($\sum_i p_i^\mu = 0$). Valid choices for the `field` keyword are multi-precision complex numbers ($\mathbb{C}$), Gaussian rationals ($\mathbb{Q}[i]$), finite fields ($\mathbb{F}_p$), and p-adic numbers ($\mathbb{Q}_p$).

```
Field("mpc", 0, 300)
Field("gaussian rational", 0, 0)
```

```
Field("finite field", 2147483647, 1)
Field("padic", 2147483629, 3)
```

Finite fields and p-adic numbers are taken from a specific slice of complex phase space, namely that with $E, p_x, p_z \in \mathbb{Q}$ and $p_y \in i\mathbb{Q}$. This is equivalent to a change of metric to $\mathrm{diag}(1, -1, 1, -1)$. Depending on the choice of field, some parameters are discarded (see Table 1).

The `Particles` class is a 1-indexed subclass of the Python built-in type `list`, with `Particle` class entries. While this re-indexing may be unusual in Python, it is the natural choice to match the notation used to write scattering amplitudes. The `Particle` objects have several attributes, each corresponding to one of the representations of the Lorentz group. For instance, we have right spinors with index up (`r_sp_u`), left spinors with index down (`l_sp_d`), rank-two spinors (`r2_sp`),

Lorentz repr.	symbol	`Particle` property	Lorentz repr.	symbol.	`Particle` property
$(0, 1/2)$	λ^α	`r_sp_u`	$(1/2, 1/2)$	p^μ	`four_mom`
	λ_α	`r_sp_d`		p_μ	`four_mom_d`
$(1/2, 0)$	$\bar{\lambda}^{\dot{\alpha}}$	`l_sp_u`		$p^{\dot{\alpha}\alpha}$	`r2_sp`
	$\bar{\lambda}_{\dot{\alpha}}$	`l_sp_d`		$\bar{p}_{\alpha\dot{\alpha}}$	`r2_sp_b`

Table 2. Representations of the Lorentz group and associated properties of the `Particle` class.

four momenta (`four_mom`), *etc.* See Table 2 for a more schematic representation. Updating the values for one of these properties automatically updates the values of the rest. The spinor conventions employed are fairly common in the field, for more details please see ref. [18, Section 2.2].

Another basic functionality provided in `lips` is the evaluation of arbitrary spinor-helicity expressions. This works seamlessly for all of the above defined fields. Understood symbols include arbitrary spinor strings, with limited support for open-index expressions, Mandelstam variables with any numbers of indices, Gram determinants of three-mass triangle diagrams (`Δ_ij|kl|mn`),[3] and traces involving γ^5 (`tr5_ijkl`). This feature can be accessed is via the `__call__` magic method of the `Particle` class, as shown.

```
Particles(5)("(8/3s23⟨24⟩[34])/(⟨15⟩⟨34⟩⟨45⟩⟨4|1+5|4])")
```

Regular expressions (with the package `re`) are used to split the string into individual invariants, which then form an abstract syntactic tree (with the package `ast`). The individual invariants are computed in the `Particles.compute` method. For simplicity's sake, greater-than and less-than symbols can be used in lieu of the angle brackets to denote holomorphic spinor contractions.

Another useful method is the `Particles.image` method, which implements transformations under the symmetries of phase space. These are represented by a tuple whose first entry is a string representing a permutation of the external legs, followed by a Boolean denoting whether a swap of the left and right representations of the Lorentz group is needed ($\lambda_\alpha \leftrightarrow \bar{\lambda}_{\dot{\alpha}}$). For instance, the symmetry of the expression in Eq. 1 would be denoted as `('12354', False)`.

3. Ideals in spinor space

Before constructing singular phase-space configurations, let us consider how to make these special configurations unambiguous. To this aim, we rely on the algebro-geometric concept of an ideal.

In `lips`, two classes are used to represent ideals: `LipsIdeal` and `SpinorIdeal`. The former represents covariant ideals in the ring of spinor components, while the latter represents invariant ideals in the ring of spinor contractions [5]. Both ideal types are subclasses of the `Ideal` class of the interface `syngular`, through which one can access many algorithms implemented in `Singular`. Despite most applications of interest deal with Lorentz invariants, it is generally convenient to work with spinor components, as the ideals are generated by fewer polynomials.

To instantiate a `LipsIdeal` object one has to declare the multiplicity of phase space, and a set of generating polynomials. For instance, taking invariants which appear in Eq. (1), we can write

```
J = LipsIdeal(5, ("<4|1+5|4]", "<5|1+4|5]"))
```

Momentum conservation is added by default. The method `make_analytical_d` of the `Particle` class is used to replace lower-index spinors with `sympy` symbols. Tensor contractions are then computed as in numeric cases. The resulting expressions are then passed to `Singular` for

[3] See Källén function or Heron's formula.

subsequent manipulations. The default ring is a polynomial ring, while the quotient ring by the momentum-conserving ideal can be accessed via the method `to_mom_cons_qring`. This modifies the ideal in place. Translation to a `SpinorIdeal`, i.e. to the Lorentz invariant subring, can be computed with the method `invariant_slice`. This relies on an elimination theory algorithm.

To understand the geometry of a variety associated to an ideal we need to compute its primary decomposition. Prime ideals will then correspond to irreducible varieties, i.e. phase space configuration where amplitudes have well-defined poles and zeros. To obtain the prime ideals associated with a given ideal, such as the above defined `J`, one can (try to) compute a primary decomposition via `Singular`. This approach is in general insufficient to map out the irreducible varieties, due to the large number of variables in the underlying ring. However, we can use physical understanding to gain insights into the primary decomposition. For instance, the ideal `J` has five branches, but only one is non-trivial.

```
K = LipsIdeal(5, ("<14>", "<15>", "<45>", "[23]" ))
L = LipsIdeal(5, ("<12>", "<13>", "<14>", "<15>", "<23>", "<24>", "<25>",
                  "<34>", "<35>", "<45>"))
M = LipsIdeal(5, ("<4|1+5|4]", "<5|1+4|5]",
                  "|1]<14><15>+|4]<14><45>-|5]<45><15>",
                  "|1>[14][15]+|4>[14][45]-|5>[45][15]"))
```

The ideals `K` and `L` are essentially triple-collinear configurations for legs 1, 4 and 5. Once these are known, the ideal `M` can be easily obtained by computing ideal quotients. To check the decomposition, we can compute an intersection of ideals with the operator `&`, which calls the command `intersect` of `Singular`, and verify the equality, via reduced Gröbner bases.

```
assert J == K & K("12345", True) & L & L("12345", True) & M
```

Note the use of another magic method (`__call__`) to compute the image of an ideal under a symmetry of phase space, similarly to the `Particles.image` method. By itself, this assertion is not sufficient to prove a primary decomposition of the ideal `J`. One also has to prove that the ideals `K`, `L`, `M` are prime. An efficient way to do this is via a prime test implement in `syngular`, which can be accessed via the method `test_primality`. This assumes equi-dimensional ideals.

4. Singular varieties

In `lips` a singular variety, irrespective of its dimension, is represented by single, generic, phase-space point—i.e. a zero-dimensional variety—embedded in the multi-dimensional variety. In this context, generic means that the result of evaluations at the chosen phase-space point should have an absolute value representative of evaluations at most points on the variety, possibly barring special, higher-codimension embedded varieties. This is guaranteed with high probability by picking the point at random, while satisfying the constraints that define the variety.

Two facilities are provided for the generation of finely-tuned phase-space points on specific varieties. First, in the submodule `hardcoded_limits` two methods, `_set` and `_set_pair`, are implemented. These methods efficiently generate points on varieties of codimension one and two respectively. However, they do not know about primary decompositions. As such, in case a variety is multi-branched, a branch will be chosen at random. Furthermore, since the constraints are solved explicitly, only some configurations can be built this way. The second option is to use the `algebraic_geometry` submodule, where the method `_singular_variety` is implemented. This method is significantly more computationally intensive than the former, as it relies on lexicographic Gröebner bases, but it allows to specify branches—i.e. irreducible sub-varieties.

For instance, the following code will generate a 3-digits 2147483647-adic phase-space point near the irreducible variety associated to the prime ideal `M`.

```
oPsM = Particles(5, field=Field("padic", 2 ** 31 - 1, 3), seed=0)
oPsM._singular_variety(("⟨4|1+5|4]", "⟨5|1+4|5]"), (1, 1),
                       generators=M.generators)
```

The first argument specifies orthogonal directions to the variety, the second the valuations of the invariants in the first argument (in this case both proportional to the chosen prime), while the `generators` keyword argument specifies the branch. Asymmetric limits can also be constructed by providing unequal valuations, see ref. [19, Appendix C].

5. Partial fraction decompositions

Partial fraction decompositions play an important role in the computation of scattering amplitudes, both in terms of final expressions, as well as at intermediate stages, e.g. for integration-by-parts identities. Standard methods are based on symbolic computations, including Gröbner bases and polynomial reduction. For instance, see ref. [20]. We can use the technology described in the previous sections to infer whether a given partial fraction decomposition is valid, before determining the analytic form of the numerator. Given denominator factors $\mathcal{D}_1$ and $\mathcal{D}_2$,

1. compute the primary decomposition for the ideal $\langle \mathcal{D}_1, \mathcal{D}_2 \rangle$;
2. generate a phase-space point near each branch of the variety $V(\langle \mathcal{D}_1, \mathcal{D}_2 \rangle)$;
3. numerically evaluate the function at these points.

If the numerator vanishes on all of them, then it belongs to (the radical of) the associated ideal (Hilbert's Nullstellensatz). For instance, given the expression of Eq. (1), we can infer that the denominator factors $\langle 4|1+5|4]$ and $\langle 5|1+4|5]$ must be separable into different fractions because the numerator, considered in least common denominator form, vanishes on all 5 branches. Further constraints from the degree of vanishing can also be imposed via the Zariski–Nagata theorem [5].

Beyond partial fractions. Partial fraction decompositions deal purely with sets of denominator factors, i.e. the poles of the functions. Yet, even if no partial fraction decomposition is possible, for instance when the denominator is a single irreducible polynomial, the numerator may still have a simple structure, generally in terms of an expanded set of invariants. These new invariants can be systematically identified via primary decompositions, and the same logic described to separate poles in the denominators can also be used to identify factors of the numerators.

References

[1] Decker W *et al.* SINGULAR 4-2-1 – A computer algebra system for polynomial computations
[2] von Manteuffel A and Schabinger R M 2015 *Phys. Lett. B* **744** 101–104 1406.4513
[3] Peraro T 2016 *JHEP* **12** 030 1608.01902
[4] De Laurentis G and Maître D 2019 *JHEP* **07** 123 1904.04067
[5] De Laurentis G and Page B 2022 *JHEP* **12** 140 2203.04269
[6] Abreu S, Page B, Pascual E and Sotnikov V 2021 *JHEP* **01** 078 2010.15834
[7] Chawdhry H A, Czakon M, Mitov A and Poncelet R 2021 *JHEP* **06** 150 2012.13553
[8] Abreu S, De Laurentis G, Ita H, Klinkert M, Page B and Sotnikov V 23xx.xxxxx
[9] Jupyter *et al.* Binder 2.0 - reproducible, interactive, sharable environments for science at scale.
[10] Harris C R *et al.* 2020 Array programming with NumPy
[11] Johansson F *et al.* 2013 mpmath: a Python library for arbitrary-precision floating-point arithmetic
[12] Meurer A *et al.* 2017 Sympy: symbolic computing in python
[13] Lenstra H *et al.* 1982 *Mathematische Annalen* **261** 515–534 URL http://eudml.org/doc/182903
[14] Monagan M 2004 pp 243–249
[15] Klappert J and Lange F 2020 *Comput. Phys. Commun.* **247** 106951 1904.00009
[16] Hostetter M 2020 Galois: A performant NumPy extension for Galois fields github.com/mhostetter/galois
[17] De Laurentis G 2021 syngular github.com/GDeLaurentis/syngular
[18] De Laurentis G *Numerical techniques for analytical high-multiplicity scattering amplitudes* Ph.D. thesis
[19] Campbell J M, De Laurentis G and Ellis R K 2022 *JHEP* **07** 096 2203.17170
[20] Heller M and von Manteuffel A 2022 *Comput. Phys. Commun.* **271** 108174 2101.08283