pySecDec Documentation

Release 0.1.1

Sophia Borowka Gudrun Heinrich Stephan Jahn
Stephen Jones Matthias Kerner
Johannes Schlenk Tom Zirke

Mar 02, 2017

CONTENTS

pySecDec is a toolbox for the calculation of dimensionally regulated parameter integrals using the sector decomposition approach *[BH00]*; see also *[Hei08]*, *[BHJ+15]*.

CHAPTER ONE

INSTALLATION

pySecDec should run fine with both, *python* 2.7 and *python* 3. It has been tested and developed on *MacOS 10.11 (El Capitan)* and *openSUSE 13.2 (Harlequin)*. However, it should be platform independent and also work on Windows.

1.1 Installation from PyPI using *pip* (coming up soon)

Installation is easiest using *pip* (https://pypi.python.org/pypi/pip). *pip* automatically installs all dependencies along with the desired package from the Python Package Index (https://pypi.python.org/pypi).

```
$ pip install pySecDec
```

1.2 Manual Installation

Before you manually install *pySecDec*, make sure that you have recent versions of *numpy* (http://www.numpy.org/) and *sympy* (http://www.sympy.org) installed.

To install *pySecDec*, open a shell in the top level directory (where `setup.py` is located) and type:

```
$ python setup.py install
```

If you have *pip*, you should type

```
$ pip install .
```

instead, for the same reasons as mentioned *above*.

1.3 The Geomethod and Normaliz

Note: If you are not urgently interested in using the `geometric decomposition`, you can ignore this section for the beginning. The instructions below are not essential for a *pySecDec* installation. You can still install normaliz **after** installing *pySecDec*. All but the `geometric decomposition` routines work without normaliz.

If you want to use the `geometric decomposition` module, you need the normaliz *[BIR]* command line executable. The `geometric decomposition` module is designed for normaliz version `3.0.0`. We recommend to set your `$PATH` such that the normaliz executable is found. Alternatively, you can pass the path to the normaliz executable directly to the functions that need it.

1.4 For Developers

pip offers an "editable" installation that can be triggered by:

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

This command causes *python* to load *pySecDec* directly from your local copy of the repository. As a result, no reinstallation is required after making changes in the source code.

pySecDec comes with a self test suite written in the *python unittest* framework. The most convenient way to run all test is using *nose* (http://nose.readthedocs.org). If *nose* is installed, just type:

```
$ nosetests
```

in the source repository to run all tests. In order to check that the examples given in the documentation are working, go to the `doc` subdirectory and type:

```
$ make doctest
```

Also note the `Makefile` in the package's root directory that implements a few common development tasks. You can list all available targets with the command

```
$ make help
```

CHAPTER TWO

OVERVIEW

pySecDec is consists of several modules that provide functions and classes for specific purposes. In this overview, we present only the most important aspects of selected modules. For detailed instruction of a specific function or class, please refer to the *reference guide*.

2.1 pySecDec.algebra

The *algebra* module implements a very basic computer algebra system. Although *sympy* in principle provides everything we need, it is way too slow for typical applications. That is because *sympy* is completely written in *python* without making use of any precompiled functions. *pySecDec*'s *algebra* module uses the in general faster *numpy* function wherever possible.

2.1.1 Polynomials

Since sector decomposition is an algorithm that acts on polynomials, we start with the key class `Polynomial`. As the name suggests, the `Polynomial` class is a container for multivariate polynomials, i.e. functions of the form:

$$\sum_i C_i \prod_j x_j^{\alpha_{ij}}$$

A multivariate polynomial is completely determined by its *coefficients* C_i, the exponents α_{ij}. The `Polynomial` class stores these in two arrays:

```
>>> from pySecDec.algebra import Polynomial
>>> poly = Polynomial([[1,0], [0,2]], ['A', 'B'])
>>> poly
 + (A)*x0 + (B)*x1**2
>>> poly.expolist
array([[1, 0],
       [0, 2]])
>>> poly.coeffs
array([A, B], dtype=object)
```

It is also possible to instatiate the `Polynomial` with by its algebraic representation:

```
>>> poly2 = Polynomial.from_expression('A*x0 + B*x1**2', ['x0','x1'])
>>> poly2
 + (A)*x0 + (B)*x1**2
>>> poly2.expolist
array([[1, 0],
       [0, 2]])
```

```
>>> poly2.coeffs
array([A, B], dtype=object)
```

Note that the second argument of *Polynomial.from_expression()* defines the variables x_j.

The *Polynomial* class implements basic operations:

```
>>> poly + 1
 + (1) + (B)*x1**2 + (A)*x0
>>> 2 * poly
 + (2*A)*x0 + (2*B)*x1**2
>>> poly + poly
 + (2*B)*x1**2 + (2*A)*x0
>>> poly * poly
 + (B**2)*x1**4 + (2*A*B)*x0*x1**2 + (A**2)*x0**2
>>> poly ** 2
 + (B**2)*x1**4 + (2*A*B)*x0*x1**2 + (A**2)*x0**2
```

2.1.2 General Expressions

In order to perform the *pySecDec.subtraction* and *pySecDec.expansion*, we have to introduce more complex algebraic constructs.

General expressions can be entered in a straightforward way:

```
>>> from pySecDec.algebra import Expression
>>> log_of_x = Expression('log(x)', ['x'])
>>> log_of_x
log( + (1)*x)
```

All expressions in the context of this *algebra* module are based on extending or combining the *Polynomials* introduced *above*. In the example above, `log_of_x` is a *LogOfPolynomial*, which is a derived class from *Polynomial*:

```
>>> type(log_of_x)
<class 'pySecDec.algebra.LogOfPolynomial'>
>>> isinstance(log_of_x, Polynomial)
True
>>> log_of_x.expolist
array([[1]])
>>> log_of_x.coeffs
array([1], dtype=object)
```

We have seen an *extension* the *Polynomial* class, now let us consider a *combination*:

```
>>> more_complex_expression = log_of_x * log_of_x
>>> more_complex_expression
(log( + (1)*x)) * (log( + (1)*x))
```

We just introduced the *Product* of two *LogOfPolynomials*:

```
>>> type(more_complex_expression)
<class 'pySecDec.algebra.Product'>
```

As suggested before, the *Product* combines two *Polynomials*. They are accessible as the `factors`:

```
>>> more_complex_expression.factors[0]
log( + (1)*x)
>>> more_complex_expression.factors[1]
log( + (1)*x)
>>> type(more_complex_expression.factors[0])
<class 'pySecDec.algebra.LogOfPolynomial'>
>>> type(more_complex_expression.factors[1])
<class 'pySecDec.algebra.LogOfPolynomial'>
```

Important: When working with this *algebra* module, it is important to understand that **everything** is based on the class `Polynomial`.

To emphasize the importance of the above statement, consider the following code:

```
>>> expression1 = Expression('x*y', ['x', 'y'])
>>> expression2 = Expression('x*y', ['x'])
>>> type(expression1)
<class 'pySecDec.algebra.Polynomial'>
>>> type(expression2)
<class 'pySecDec.algebra.Polynomial'>
>>> expression1
 + (1)*x*y
>>> expression2
 + (y)*x
```

Although `expression1` and `expression2` are mathematically identical, they are treated differently by the *algebra* module. In `expression1`, both, `x` and `y`, are considered as variables of the `Polynomial`. In contrast, `y` is treated as *coefficient* in `expression2`:

```
>>> expression1.expolist
array([[1, 1]])
>>> expression1.coeffs
array([1], dtype=object)
>>> expression2.expolist
array([[1]])
>>> expression2.coeffs
array([y], dtype=object)
```

The second argument of the function `Expression` controls how the variables are distributed between the coefficients and the variables in the underlying `Polynomials`. Keep that in mind in order to avoid confusion. One can always check which symbols are considered as variables by asking for the `symbols`:

```
>>> expression1.symbols
[x, y]
>>> expression2.symbols
[x]
```

2.2 Feynman Parametrization of Loop Integrals

The primary purpose of *pySecDec* is calculating loop integrals as they arise in fixed order calculations in quantum field theories. In our approach, the first step is converting the loop integral from the momentum representation to the Feynman parameter representation.

The module *pySecDec.loop_integral* implements exactly that conversion. The most basic use is to calculate the first `U` and the second `F` Symanzik polynomial from the propagators of a loop integral.

2.2.1 One Loop Bubble

To calculate `U` and `F` of the one loop bubble, type the following commands:

```
>>> from pySecDec.loop_integral import LoopIntegralFromPropagators
>>> propagators = ['k**2', '(k - p)**2']
>>> loop_momenta = ['k']
>>> one_loop_bubble = LoopIntegralFromPropagators(propagators, loop_momenta)
>>> one_loop_bubble.U
 + (1)*x0 + (1)*x1
>>> one_loop_bubble.F
 + (-p**2)*x0*x1
```

The example above among other useful features is also stated in the full documenation of *LoopIntegralFromPropagators()* in the reference guide.

2.2.2 Two Loop Planar Box with Numerator

Consider the propagators of the two loop planar box:

```
>>> propagators = ['k1**2','(k1+p2)**2',
...                '(k1-p1)**2','(k1-k2)**2',
...                '(k2+p2)**2','(k2-p1)**2',
...                '(k2+p2+p3)**2']
>>> loop_momenta = ['k1','k2']
```

We could now instantiate the *LoopIntegral* just like *before*. However, let us consider an additional numerator:

```
>>> numerator = 'k1(mu)*k1(mu) + 2*k1(mu)*p3(mu) + p3(mu)*p3(mu)' # (k1 + p3) ** 2
```

In order to unambiguously define the loop integral, we must state which symbols denote the `Lorentz_indices` (just `mu` in this case here) and the `external_momenta`:

```
>>> external_momenta = ['p1','p2','p3','p4']
>>> Lorentz_indices=['mu']
```

With that, we can Feynman parametrize the two loop box with a numerator:

```
>>> box = LoopIntegralFromPropagators(propagators, loop_momenta, external_momenta,
...                                   numerator=numerator, Lorentz_indices=Lorentz_
↪indices)
>>> box.U
 + (1)*x3*x6 + (1)*x3*x5 + (1)*x3*x4 + (1)*x2*x6 + (1)*x2*x5 + (1)*x2*x4 + (1)*x2*x3
↪+ (1)*x1*x6 + (1)*x1*x5 + (1)*x1*x4 + (1)*x1*x3 + (1)*x0*x6 + (1)*x0*x5 + (1)*x0*x4
↪+ (1)*x0*x3
>>> box.F
 + (-p1**2 - 2*p1*p2 - 2*p1*p3 - p2**2 - 2*p2*p3 - p3**2)*x3*x5*x6 + (-
↪p3**2)*x3*x4*x6 + (-p1**2 - 2*p1*p2 - p2**2)*x3*x4*x5 + (-p1**2 - 2*p1*p2 - 2*p1*p3
↪- p2**2 - 2*p2*p3 - p3**2)*x2*x5*x6 + (-p3**2)*x2*x4*x6 + (-p1**2 - 2*p1*p2 -
↪p2**2)*x2*x4*x5 + (-p1**2 - 2*p1*p2 - 2*p1*p3 - p2**2 - 2*p2*p3 - p3**2)*x2*x3*x6 +
↪(-p1**2 - 2*p1*p2 - p2**2)*x2*x3*x4 + (-p1**2 - 2*p1*p2 - 2*p1*p3 - p2**2 - 2*p2*p3
↪- p3**2)*x1*x5*x6 + (-p3**2)*x1*x4*x6 + (-p1**2 - 2*p1*p2 - p2**2)*x1*x4*x5 + (-
↪p3**2)*x1*x3*x6 + (-p1**2 - 2*p1*p2 - p2**2)*x1*x3*x5 + (-p1**2 - 2*p1*p2 -
↪p2**2)*x1*x2*x6 + (-p1**2 - 2*p1*p2 - p2**2)*x1*x2*x5 + (-p1**2 - 2*p1*p2 -
↪p2**2)*x1*x2*x4 + (-p1**2 - 2*p1*p2 - p2**2)*x1*x2*x3 + (-p1**2 - 2*p1*p2 - 2*p1*p3
↪- p2**2 - 2*p2*p3 - p3**2)*x0*x5*x6 + (-p3**2)*x0*x4*x6 + (-p1**2 - 2*p1*p2 -
↪p2**2)*x0*x4*x5 + (-p2**2 - 2*p2*p3 - p3**2)*x0*x3*x6 + (-p1**2)*x0*x3*x5 + (-
↪p2**2)*x0*x3*x4 + (-p1**2)*x0*x2*x6 + (-p1**2)*x0*x2*x5 + (-p1**2)*x0*x2*x4 + (-
↪p1**2)*x0*x2*x3 + (-p2**2)*x0*x1*x6 + (-p2**2)*x0*x1*x5 + (-p2**2)*x0*x1*x4 + (-
↪p2**2)*x0*x1*x3
```

```
>>> box.numerator
 + (-2*eps*p3(mu)**2 - 2*p3(mu)**2)*U**2 + (-eps + 2)*x6*F + (-eps + 2)*x5*F + (-eps␣
↪+ 2)*x4*F + (-eps + 2)*x3*F + (4*eps*p2(mu)*p3(mu) + 4*eps*p3(mu)**2 +␣
↪4*p2(mu)*p3(mu) + 4*p3(mu)**2)*x3*x6*U + (-4*eps*p1(mu)*p3(mu) -␣
↪4*p1(mu)*p3(mu))*x3*x5*U + (4*eps*p2(mu)*p3(mu) + 4*p2(mu)*p3(mu))*x3*x4*U + (-
↪2*eps*p2(mu)**2 - 4*eps*p2(mu)*p3(mu) - 2*eps*p3(mu)**2 - 2*p2(mu)**2 -␣
↪4*p2(mu)*p3(mu) - 2*p3(mu)**2)*x3**2*x6**2 + (4*eps*p1(mu)*p2(mu) +␣
↪4*eps*p1(mu)*p3(mu) + 4*p1(mu)*p2(mu) + 4*p1(mu)*p3(mu))*x3**2*x5*x6 + (-
↪2*eps*p1(mu)**2 - 2*p1(mu)**2)*x3**2*x5**2 + (-4*eps*p2(mu)**2 -␣
↪4*eps*p2(mu)*p3(mu) - 4*p2(mu)**2 - 4*p2(mu)*p3(mu))*x3**2*x4*x6 +␣
↪(4*eps*p1(mu)*p2(mu) + 4*p1(mu)*p2(mu))*x3**2*x4*x5 + (-2*eps*p2(mu)**2 -␣
↪2*p2(mu)**2)*x3**2*x4**2 + (-4*eps*p1(mu)*p3(mu) - 4*p1(mu)*p3(mu))*x2*x6*U + (-
↪4*eps*p1(mu)*p3(mu) - 4*p1(mu)*p3(mu))*x2*x5*U + (-4*eps*p1(mu)*p3(mu) -␣
↪4*p1(mu)*p3(mu))*x2*x4*U + (-4*eps*p1(mu)*p3(mu) - 4*p1(mu)*p3(mu))*x2*x3*U +␣
↪(4*eps*p1(mu)*p2(mu) + 4*eps*p1(mu)*p3(mu) + 4*p1(mu)*p2(mu) +␣
↪4*p1(mu)*p3(mu))*x2*x3*x6**2 + (-4*eps*p1(mu)**2 + 4*eps*p1(mu)*p2(mu) +␣
↪4*eps*p1(mu)*p3(mu) - 4*p1(mu)**2 + 4*p1(mu)*p2(mu) + 4*p1(mu)*p3(mu))*x2*x3*x5*x6␣
↪+ (-4*eps*p1(mu)**2 - 4*p1(mu)**2)*x2*x3*x5**2 + (8*eps*p1(mu)*p2(mu) +␣
↪4*eps*p1(mu)*p3(mu) + 8*p1(mu)*p2(mu) + 4*p1(mu)*p3(mu))*x2*x3*x4*x6 + (-
↪4*eps*p1(mu)**2 + 4*eps*p1(mu)*p2(mu) - 4*p1(mu)**2 + 4*p1(mu)*p2(mu))*x2*x3*x4*x5␣
↪+ (4*eps*p1(mu)*p2(mu) + 4*p1(mu)*p2(mu))*x2*x3*x4**2 + (4*eps*p1(mu)*p2(mu) +␣
↪4*eps*p1(mu)*p3(mu) + 4*p1(mu)*p2(mu) + 4*p1(mu)*p3(mu))*x2*x3**2*x6 + (-
↪4*eps*p1(mu)**2 - 4*p1(mu)**2)*x2*x3**2*x5 + (4*eps*p1(mu)*p2(mu) +␣
↪4*p1(mu)*p2(mu))*x2*x3**2*x4 + (-2*eps*p1(mu)**2 - 2*p1(mu)**2)*x2**2*x6**2 + (-
↪4*eps*p1(mu)**2 - 4*p1(mu)**2)*x2**2*x5*x6 + (-2*eps*p1(mu)**2 -␣
↪2*p1(mu)**2)*x2**2*x5**2 + (-4*eps*p1(mu)**2 - 4*p1(mu)**2)*x2**2*x4*x6 + (-
↪4*eps*p1(mu)**2 - 4*p1(mu)**2)*x2**2*x4*x5 + (-2*eps*p1(mu)**2 -␣
↪2*p1(mu)**2)*x2**2*x4**2 + (-4*eps*p1(mu)**2 - 4*p1(mu)**2)*x2**2*x3*x6 + (-
↪4*eps*p1(mu)**2 - 4*p1(mu)**2)*x2**2*x3*x5 + (-4*eps*p1(mu)**2 -␣
↪4*p1(mu)**2)*x2**2*x3*x4 + (-2*eps*p1(mu)**2 - 2*p1(mu)**2)*x2**2*x3**2 +␣
↪(4*eps*p2(mu)*p3(mu) + 4*p2(mu)*p3(mu))*x1*x6*U + (4*eps*p2(mu)*p3(mu) +␣
↪4*p2(mu)*p3(mu))*x1*x5*U + (4*eps*p2(mu)*p3(mu) + 4*p2(mu)*p3(mu))*x1*x4*U +␣
↪(4*eps*p2(mu)*p3(mu) + 4*p2(mu)*p3(mu))*x1*x3*U + (-4*eps*p2(mu)**2 -␣
↪4*eps*p2(mu)*p3(mu) - 4*p2(mu)**2 - 4*p2(mu)*p3(mu))*x1*x3*x6**2 +␣
↪(4*eps*p1(mu)*p2(mu) - 4*eps*p2(mu)**2 - 4*eps*p2(mu)*p3(mu) + 4*p1(mu)*p2(mu) -␣
↪4*p2(mu)**2 - 4*p2(mu)*p3(mu))*x1*x3*x5*x6 + (4*eps*p1(mu)*p2(mu) +␣
↪4*p1(mu)*p2(mu))*x1*x3*x5**2 + (-8*eps*p2(mu)**2 - 4*eps*p2(mu)*p3(mu) -␣
↪8*p2(mu)**2 - 4*p2(mu)*p3(mu))*x1*x3*x4*x6 + (4*eps*p1(mu)*p2(mu) - 4*eps*p2(mu)**2␣
↪+ 4*p1(mu)*p2(mu) - 4*p2(mu)**2)*x1*x3*x4*x5 + (-4*eps*p2(mu)**2 -␣
↪4*p2(mu)**2)*x1*x3*x4**2 + (-4*eps*p2(mu)**2 - 4*eps*p2(mu)*p3(mu) - 4*p2(mu)**2 -␣
↪4*p2(mu)*p3(mu))*x1*x3**2*x6 + (4*eps*p1(mu)*p2(mu) + 4*p1(mu)*p2(mu))*x1*x3**2*x5␣
↪+ (-4*eps*p2(mu)**2 - 4*p2(mu)**2)*x1*x3**2*x4 + (4*eps*p1(mu)*p2(mu) +␣
↪4*p1(mu)*p2(mu))*x1*x2*x6**2 + (8*eps*p1(mu)*p2(mu) + 8*p1(mu)*p2(mu))*x1*x2*x5*x6␣
↪+ (4*eps*p1(mu)*p2(mu) + 4*p1(mu)*p2(mu))*x1*x2*x5**2 + (8*eps*p1(mu)*p2(mu) +␣
↪8*p1(mu)*p2(mu))*x1*x2*x4*x6 + (8*eps*p1(mu)*p2(mu) + 8*p1(mu)*p2(mu))*x1*x2*x4*x5␣
↪+ (4*eps*p1(mu)*p2(mu) + 4*p1(mu)*p2(mu))*x1*x2*x4**2 + (8*eps*p1(mu)*p2(mu) +␣
↪8*p1(mu)*p2(mu))*x1*x2*x3*x6 + (8*eps*p1(mu)*p2(mu) + 8*p1(mu)*p2(mu))*x1*x2*x3*x5␣
↪+ (8*eps*p1(mu)*p2(mu) + 8*p1(mu)*p2(mu))*x1*x2*x3*x4 + (4*eps*p1(mu)*p2(mu) +␣
↪4*p1(mu)*p2(mu))*x1*x2*x3**2 + (-2*eps*p2(mu)**2 - 2*p2(mu)**2)*x1**2*x6**2 + (-
↪4*eps*p2(mu)**2 - 4*p2(mu)**2)*x1**2*x5*x6 + (-2*eps*p2(mu)**2 -␣
↪2*p2(mu)**2)*x1**2*x5**2 + (-4*eps*p2(mu)**2 - 4*p2(mu)**2)*x1**2*x4*x6 + (-
↪4*eps*p2(mu)**2 - 4*p2(mu)**2)*x1**2*x4*x5 + (-2*eps*p2(mu)**2 -␣
↪2*p2(mu)**2)*x1**2*x4**2 + (-4*eps*p2(mu)**2 - 4*p2(mu)**2)*x1**2*x3*x6 + (-
↪4*eps*p2(mu)**2 - 4*p2(mu)**2)*x1**2*x3*x5 + (-4*eps*p2(mu)**2 -␣
↪4*p2(mu)**2)*x1**2*x3*x4 + (-2*eps*p2(mu)**2 - 2*p2(mu)**2)*x1**2*x3**2
```

We can also generate output in terms of Mandelstam invariants:

```
>>> replacement_rules = [
...                         ('p1*p1', 0),
...                         ('p2*p2', 0),
...                         ('p3*p3', 0),
...                         ('p4*p4', 0),
...                         ('p1*p2', 's/2'),
...                         ('p2*p3', 't/2'),
...                         ('p1*p3', '-s/2-t/2')
...                     ]
>>> box = LoopIntegralFromPropagators(propagators, loop_momenta, external_momenta,
...                                   numerator=numerator, Lorentz_indices=Lorentz_
↪indices,
...                                   replacement_rules=replacement_rules)
>>> box.U
 + (1)*x3*x6 + (1)*x3*x5 + (1)*x3*x4 + (1)*x2*x6 + (1)*x2*x5 + (1)*x2*x4 + (1)*x2*x3␣
↪+ (1)*x1*x6 + (1)*x1*x5 + (1)*x1*x4 + (1)*x1*x3 + (1)*x0*x6 + (1)*x0*x5 + (1)*x0*x4␣
↪+ (1)*x0*x3
>>> box.F
 + (-s)*x3*x4*x5 + (-s)*x2*x4*x5 + (-s)*x2*x3*x4 + (-s)*x1*x4*x5 + (-s)*x1*x3*x5 + (-
↪s)*x1*x2*x6 + (-s)*x1*x2*x5 + (-s)*x1*x2*x4 + (-s)*x1*x2*x3 + (-s)*x0*x4*x5 + (-
↪t)*x0*x3*x6
>>> box.numerator
 + (-eps + 2)*x6*F + (-eps + 2)*x5*F + (-eps + 2)*x4*F + (-eps + 2)*x3*F + (2*eps*t +␣
↪2*t)*x3*x6*U + (-4*eps*(-s/2 - t/2) + 2*s + 2*t)*x3*x5*U + (2*eps*t + 2*t)*x3*x4*U␣
↪+ (-2*eps*t - 2*t)*x3**2*x6**2 + (2*eps*s + 4*eps*(-s/2 - t/2) - 2*t)*x3**2*x5*x6 +␣
↪(-2*eps*t - 2*t)*x3**2*x4*x6 + (2*eps*s + 2*s)*x3**2*x4*x5 + (-4*eps*(-s/2 - t/2) +␣
↪2*s + 2*t)*x2*x6*U + (-4*eps*(-s/2 - t/2) + 2*s + 2*t)*x2*x5*U + (-4*eps*(-s/2 - t/
↪2) + 2*s + 2*t)*x2*x4*U + (-4*eps*(-s/2 - t/2) + 2*s + 2*t)*x2*x3*U + (2*eps*s +␣
↪4*eps*(-s/2 - t/2) - 2*t)*x2*x3*x6**2 + (2*eps*s + 4*eps*(-s/2 - t/2) -␣
↪2*t)*x2*x3*x5*x6 + (4*eps*s + 4*eps*(-s/2 - t/2) + 2*s - 2*t)*x2*x3*x4*x6 +␣
↪(2*eps*s + 2*s)*x2*x3*x4*x5 + (2*eps*s + 2*s)*x2*x3*x4**2 + (2*eps*s + 4*eps*(-s/2 -
↪ t/2) - 2*t)*x2*x3**2*x6 + (2*eps*s + 2*s)*x2*x3**2*x4 + (2*eps*t + 2*t)*x1*x6*U +␣
↪(2*eps*t + 2*t)*x1*x5*U + (2*eps*t + 2*t)*x1*x4*U + (2*eps*t + 2*t)*x1*x3*U + (-
↪2*eps*t - 2*t)*x1*x3*x6**2 + (2*eps*s - 2*eps*t + 2*s - 2*t)*x1*x3*x5*x6 + (2*eps*s␣
↪+ 2*s)*x1*x3*x5**2 + (-2*eps*t - 2*t)*x1*x3*x4*x6 + (2*eps*s + 2*s)*x1*x3*x4*x5 + (-
↪2*eps*t - 2*t)*x1*x3**2*x6 + (2*eps*s + 2*s)*x1*x3**2*x5 + (2*eps*s +␣
↪2*s)*x1*x2*x6**2 + (4*eps*s + 4*s)*x1*x2*x5*x6 + (2*eps*s + 2*s)*x1*x2*x5**2 +␣
↪(4*eps*s + 4*s)*x1*x2*x4*x6 + (4*eps*s + 4*s)*x1*x2*x4*x5 + (2*eps*s +␣
↪2*s)*x1*x2*x4**2 + (4*eps*s + 4*s)*x1*x2*x3*x6 + (4*eps*s + 4*s)*x1*x2*x3*x5 +␣
↪(4*eps*s + 4*s)*x1*x2*x3*x4 + (2*eps*s + 2*s)*x1*x2*x3**2
```

2.3 Sector Decomposition

The sector decomposition algorithm aims to factorize the polynomials P_i as products of a monomial and a polynomial with nonzero constant term:

$$P_i(\{x_j\}) \longmapsto \prod_j x_j^{\alpha_j} \left(const + p_i(\{x_j\})\right).$$

Factorizing polynomials in that way by expoliting integral transformations is the first step in an algorithm for solving dimensionally regulated integrals of the form

$$\int_0^1 \prod_{i,j} P_i(\{x_j\})^{\beta_i} \, dx_j.$$

The iterative sector decomposition splits the integral and remaps the integration domain until all polynomials P_i in all arising integrals (called *sectors*) have the desired form $const + polynomial$. An introduction to the sector decomposition approach can be found in *[Hei08]*.

To demonstrate the `pySecDec.decomposition` module, we decompose the polynomials

```
>>> p1 = Polynomial.from_expression('x + A*y', ['x','y','z'])
>>> p2 = Polynomial.from_expression('x + B*y*z', ['x','y','z'])
```

Let us first focus on the iterative decomposition of `p1`. In the *pySecDec* framework, we first have to pack `p1` into a `Sector`:

```
>>> from pySecDec.decomposition import Sector
>>> initial_sector = Sector([p1])
>>> print(initial_sector)
Sector:
Jacobian= + (1)
cast=[( + (1)) * ( + (1)*x + (A)*y)]
other=[]
```

We can now run the iterative decomposition and take a look at the decomposed sectors:

```
>>> from pySecDec.decomposition.iterative import iterative_decomposition
>>> decomposed_sectors = iterative_decomposition(initial_sector)
>>> for sector in decomposed_sectors:
...     print(sector)
...     print('\n')
...
Sector:
Jacobian= + (1)*x
cast=[( + (1)*x) * ( + (1) + (A)*y)]
other=[]


Sector:
Jacobian= + (1)*y
cast=[( + (1)*y) * ( + (1)*x + (A))]
other=[]
```

The decomposition of `p2` needs two iterations and yields three sectors:

```
>>> initial_sector = Sector([p2])
>>> decomposed_sectors = iterative_decomposition(initial_sector)
>>> for sector in decomposed_sectors:
...     print(sector)
...     print('\n')
...
Sector:
Jacobian= + (1)*x
cast=[( + (1)*x) * ( + (1) + (B)*y*z)]
other=[]


Sector:
Jacobian= + (1)*x*y
cast=[( + (1)*x*y) * ( + (1) + (B)*z)]
other=[]
```

```
Sector:
Jacobian= + (1)*y*z
cast=[( + (1)*y*z) * ( + (1)*x + (B))]
other=[]
```

Note that we declared z as variable for p1 although it does not depend on it. However, we have to do so if we want to simultaneously decompose p1 and p2:

```
>>> initial_sector = Sector([p1, p2])
>>> decomposed_sectors = iterative_decomposition(initial_sector)
>>> for sector in decomposed_sectors:
...     print(sector)
...     print('\n')
...
Sector:
Jacobian= + (1)*x
cast=[( + (1)*x) * ( + (1) + (A)*y), ( + (1)*x) * ( + (1) + (B)*y*z)]
other=[]


Sector:
Jacobian= + (1)*x*y
cast=[( + (1)*y) * ( + (1)*x + (A)), ( + (1)*x*y) * ( + (1) + (B)*z)]
other=[]


Sector:
Jacobian= + (1)*y*z
cast=[( + (1)*y) * ( + (1)*x*z + (A)), ( + (1)*y*z) * ( + (1)*x + (B))]
other=[]
```

We just fully decomposed p1 and p2. In some cases, one may want to bring one polynomial, say p1, into standard form, but not neccessarily the other. For that purpose, the *Sector* can take a second argument. In the following code example, we bring p1 into standard form, apply all transformations to p2 as well, but stop before p2 is fully decomposed:

```
>>> initial_sector = Sector([p1], [p2])
>>> decomposed_sectors = iterative_decomposition(initial_sector)
>>> for sector in decomposed_sectors:
...     print(sector)
...     print('\n')
...
Sector:
Jacobian= + (1)*x
cast=[( + (1)*x) * ( + (1) + (A)*y)]
other=[ + (1)*x + (B)*x*y*z]


Sector:
Jacobian= + (1)*y
cast=[( + (1)*y) * ( + (1)*x + (A))]
other=[ + (1)*x*y + (B)*y*z]
```

2.4 Expansion

The purpose of the *expansion* module is, as the name suggests, to provide routines to perform a series expansion. The module basically implements two routines - the Taylor expansion (*pySecDec.expansion.expand_Taylor()*) and an expansion of polyrational functions supporting singularities in the expansion variable (*pySecDec.expansion.expand_singular()*).

2.4.1 Taylor expansion

The function *pySecDec.expansion.expand_Taylor()* implements the ordinary Taylor expansion. It takes an algebraic expression (in the sense of the *algebra module*, the index of the expansion variable and the order to which the expression shall be expanded:

```
>>> from pySecDec.algebra import Expression
>>> from pySecDec.expansion import expand_Taylor
>>> expression = Expression('x**eps', ['eps'])
>>> expand_Taylor(expression, 0, 2).simplify()
 + (1) + (log( + (x)))*eps + ((log( + (x))) * (log( + (x))) * ( + (1/2)))*eps**2
```

It is also possible to expand an expression in multiple variables simultaneously:

```
>>> expression = Expression('x**(eps + alpha)', ['eps', 'alpha'])
>>> expand_Taylor(expression, [0,1], [2,0]).simplify()
 + (1) + (log( + (x)))*eps + ((log( + (x))) * (log( + (x))) * ( + (1/2)))*eps**2
```

The command above instructs *pySecDec.expansion.expand_Taylor()* to expand the `expression` to the second order in the variable indexed `0` (`eps`) and to the zeroth order in the variable indexed `1` (`alpha`).

2.4.2 Laurent Expansion

pySecDec.expansion.expand_singular() Laurent expands polyrational functions.

Its input is more restrictive than for the *Taylor expansion*. It expects a *Product* where the factors are either *Polynomials* or *ExponentiatedPolynomials* with `exponent = -1`:

```
>>> from pySecDec.expansion import expand_singular
>>> expression = Expression('1/(eps + alpha)', ['eps', 'alpha']).simplify()
>>> expand_singular(expression, 0, 1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/pcl340a/sjahn/Projects/pySecDec/pySecDec/expansion.py", line 241, in␣
↪expand_singular
    return _expand_and_flatten(product, indices, orders, _expand_singular_step)
  File "/home/pcl340a/sjahn/Projects/pySecDec/pySecDec/expansion.py", line 209, in _
↪expand_and_flatten
    expansion = recursive_expansion(expression, indices, orders)
  File "/home/pcl340a/sjahn/Projects/pySecDec/pySecDec/expansion.py", line 198, in␣
↪recursive_expansion
    expansion = expansion_one_variable(expression, index, order)
  File "/home/pcl340a/sjahn/Projects/pySecDec/pySecDec/expansion.py", line 82, in _
↪expand_singular_step
    raise TypeError('`product` must be a `Product`')
TypeError: `product` must be a `Product`
>>> expression # ``expression`` is indeed a polyrational function.
( + (1)*alpha + (1)*eps)**(-1)
```

```
>>> type(expression) # It is just not packed in a ``Product`` as ``expand_singular``␣
↪expects.
<class 'pySecDec.algebra.ExponentiatedPolynomial'>
>>> from pySecDec.algebra import Product
>>> expression = Product(expression)
>>> expand_singular(expression, 0, 1)
 + (( + (1)) * (( + (1)*alpha)**(-1))) + (( + (-1)) * (( + (1)*alpha**2)**(-1)))*eps
```

Like in the *Taylor expansion*, we can expand simultaneously in multiple parameters. Note, however, that the result of the Laurent expansion depends on the ordering of the expansion variables. The second argument of `pySecDec.expansion.expand_singular()` determines the order of the expansion:

```
>>> expression = Expression('1/(2*eps) * 1/(eps + alpha)', ['eps', 'alpha']).
↪simplify()
>>> eps_first = expand_singular(expression, [0,1], [1,1])
>>> eps_first
 + ( + (( + (1/2)) * (( + (1))**(-1)))*alpha**-1)*eps**-1 + ( + (( + (-1/2)) * (( +␣
↪(1))**(-1)))*alpha**-2) + ( + (( + (1)) * (( + (2))**(-1)))*alpha**-3)*eps
>>> alpha_first = expand_singular(expression, [1,0], [1,1])
>>> alpha_first
 + ( + (( + (1/2)) * (( + (1))**(-1)))*eps**-2) + ( + (( + (-1/2)) * (( + (1))**(-
↪1)))*eps**-3)*alpha
```

The expression printed out by our algebra module are quite messy. In order to obtain nicer output, we can convert these expressions to the slower but more high level *sympy*:

```
>>> import sympy as sp
>>> eps_first = expand_singular(expression, [0,1], [1,1])
>>> alpha_first = expand_singular(expression, [1,0], [1,1])
>>> sp.sympify(eps_first)
1/(2*alpha*eps) - 1/(2*alpha**2) + eps/(2*alpha**3)
>>> sp.sympify(alpha_first)
-alpha/(2*eps**3) + 1/(2*eps**2)
```

CHAPTER

THREE

REFERENCE GUIDE

Implementation of a simple computer algebra system

class pySecDec.algebra.**DerivativeTracker**(*expression*, *copy=False*)

Keep track of all derivatives taken of an _Expression. When the derive() method is called, save the multiindex of the derivative to be taken.

Parameters

- **expression** – _Expression in the sense of this module; The *expression* to track the derivatives.
- **copy** – bool; Whether or not to copy the *expression*.

The derivative multiindices are the keys in the *dictionary* self.derivatives. The values are lists with two elements: Its first element is the index to derive the derivative indicated by the multiindex in the second element by, in order to abtain the derivative indicated by the key:

```
>>> from pySecDec.algebra import Polynomial, DerivativeTracker
>>> poly = Polynomial.from_expression('x**2*y + y**2', ['x','y'])
>>> tracker = DerivativeTracker(poly)
>>> tracker.derive(0).derive(1)
DerivativeTracker( + (2)*x, index = (1, 1), derivatives = {(1, 0): [0, (0, 0)],
↪(1, 1): [1, (1, 0)]})
>>> tracker.derivatives
{(1, 0): [0, (0, 0)], (1, 1): [1, (1, 0)]}
```

copy()

Return a copy of a DerivativeTracker.

derive(*index*)

Generate the derivative of the *expression* and update self.derivatives.

replace(*index*, *value*, *remove=False*)

Replace a variable in an expression by a number or a symbol. The entries in all expolist of the underlying Polynomial are set to zero. The coefficients are modified according to *value* and the powers indicated in the expolist.

Parameters

- **expression** – _Expression; The expression to replace the variable.
- **index** – integer; The index of the variable to be replaced.
- **value** – number or sympy expression; The value to insert for the chosen variable.
- **remove** – bool; Whether or not to remove the replaced parameter from the parameters in the *expression*.

simplify()
Simplify the *expression*.

class pySecDec.algebra.**ExponentiatedPolynomial**(*expolist*, *coeffs*, *exponent=1*, *polysymbols='x'*, *copy=True*)
Like *Polynomial*, but with a global exponent. $polynomial^{exponent}$

Parameters

- **expolist** – iterable of iterables; The variable's powers for each term.
- **coeffs** – iterable; The coefficients of the polynomial.
- **exponent** – object, optional; The global exponent.
- **polysymbols** – iterable or string, optional; The symbols to be used for the polynomial variables when converted to string. If a string is passed, the variables will be consecutively numbered.

  For example: expolist=[[2,0],[1,1]] coeffs=["A","B"]

  – polysymbols='x' (default) <-> "A*x0**2 + B*x0*x1"

  – polysymbols=['x','y'] <-> "A*x**2 + B*x*y"
- **copy** – bool; Whether or not to copy the *expolist*, the *coeffs*, and the *exponent*.

Note: If copy is False, it is assumed that the *expolist*, the *coeffs* and the *exponent* have the correct type.

copy()
Return a copy of a *Polynomial* or a subclass.

derive(*index*)
Generate the derivative by the parameter indexed *index*.

Parameters **index** – integer; The index of the paramater to derive by.

simplify()
Apply the identity <something>**0 = 1 or <something>**1 = <somethng> if possible, otherwise call the simplify method of the base class. Convert exponent to symbol if possible.

pySecDec.algebra.**Expression**(*expression*, *polysymbols*)
Convert a sympy expression to an expression in terms of this module.

Parameters

- **expression** – string or sympy expression; The expression to be converted
- **polysymbols** – iterable of strings or sympy symbols; The symbols to be stored as expolists (see *Polynomial*) where possible.

class pySecDec.algebra.**Function**(*symbol*, **arguments*, ***kwargs*)
Symbolic function that can take care of parameter transformations.

Parameters

- **symbol** – string; The symbol to be used to represent the *Function*.
- **arguments** – arbitrarily many _Expression; The arguments of the *Function*.
- **copy** – bool; Whether or not to copy the *arguments*.

copy()
Return a copy of a `Function`.

derive(*index*)
Generate the derivative by the parameter indexed *index*. The derivative of a function with *symbol* `f` by some *index* is denoted as `df_d<index>`.

Parameters **index** – integer; The index of the paramater to derive by.

replace(*expression*, *index*, *value*, *remove=False*)
Replace a variable in an expression by a number or a symbol. The entries in all `expolist` of the underlying `Polynomial` are set to zero. The coefficients are modified according to *value* and the powers indicated in the `expolist`.

Parameters

- **expression** – `_Expression`; The expression to replace the variable.
- **index** – integer; The index of the variable to be replaced.
- **value** – number or sympy expression; The value to insert for the chosen variable.
- **remove** – bool; Whether or not to remove the replaced parameter from the `parameters` in the *expression*.

simplify()
Simplify the arguments.

class `pySecDec.algebra.`**Log**(*arg*, *copy=True*)
The (natural) logarithm to base e (2.718281828459..). Store the expressions `log(arg)`.

Parameters

- **arg** – `_Expression`; The argument of the logarithm.
- **copy** – bool; Whether or not to copy the *arg*.

copy()
Return a copy of a `Log`.

derive(*index*)
Generate the derivative by the parameter indexed *index*.

Parameters **index** – integer; The index of the paramater to derive by.

replace(*expression*, *index*, *value*, *remove=False*)
Replace a variable in an expression by a number or a symbol. The entries in all `expolist` of the underlying `Polynomial` are set to zero. The coefficients are modified according to *value* and the powers indicated in the `expolist`.

Parameters

- **expression** – `_Expression`; The expression to replace the variable.
- **index** – integer; The index of the variable to be replaced.
- **value** – number or sympy expression; The value to insert for the chosen variable.
- **remove** – bool; Whether or not to remove the replaced parameter from the `parameters` in the *expression*.

simplify()
Apply `log(1) = 0`.

class `pySecDec.algebra.`**LogOfPolynomial**(*expolist*, *coeffs*, *polysymbols='x'*, *copy=True*)
The natural logarithm of a `Polynomial`.

Parameters

- **`expolist`** – iterable of iterables; The variable's powers for each term.
- **`coeffs`** – iterable; The coefficients of the polynomial.
- **`exponent`** – object, optional; The global exponent.
- **`polysymbols`** – iterable or string, optional; The symbols to be used for the polynomial variables when converted to string. If a string is passed, the variables will be consecutively numbered.

  For example: expolist=[[2,0],[1,1]] coeffs=["A","B"]

  – polysymbols='x' (default) <-> "A*x0**2 + B*x0*x1"

  – polysymbols=['x','y'] <-> "A*x**2 + B*x*y"

`derive`(*index*)

Generate the derivative by the parameter indexed *index*.

Parameters **`index`** – integer; The index of the paramater to derive by.

static `from_expression`(*expression*, *polysymbols*)

Alternative constructor. Construct the *`LogOfPolynomial`* from an algebraic expression.

Parameters

- **`expression`** – string or sympy expression; The algebraic representation of the polynomial, e.g. "5*x1**2 + x1*x2"
- **`polysymbols`** – iterable of strings or sympy symbols; The symbols to be interpreted as the polynomial variables, e.g. "['x1','x2']".

`simplify`()

Apply the identity `log(1) = 0`, otherwise call the simplify method of the base class.

class `pySecDec.algebra.`**`Polynomial`**(*expolist*, *coeffs*, *polysymbols='x'*, *copy=True*)

Container class for polynomials. Store a polynomial as list of lists counting the powers of the variables. For example the polynomial "x1**2 + x1*x2" is stored as [[2,0],[1,1]].

Coefficients are stored in a separate list of strings, e.g. "A*x0**2 + B*x0*x1" <-> [[2,0],[1,1]] and ["A","B"].

Parameters

- **`expolist`** – iterable of iterables; The variable's powers for each term.

  Hint: Negative powers are allowed.

- **`coeffs`** – 1d array-like with numerical or sympy-symbolic (see http://www.sympy.org/) content, e.g. [x,1,2] where x is a sympy symbol; The coefficients of the polynomial.
- **`polysymbols`** – iterable or string, optional; The symbols to be used for the polynomial variables when converted to string. If a string is passed, the variables will be consecutively numbered.

  For example: expolist=[[2,0],[1,1]] coeffs=["A","B"]

  – polysymbols='x' (default) <-> "A*x0**2 + B*x0*x1"

  – polysymbols=['x','y'] <-> "A*x**2 + B*x*y"

- **copy** – bool; Whether or not to copy the *expolist* and the *coeffs*.

> **Note:** If copy is `False`, it is assumed that the *expolist* and the *coeffs* have the correct type.

becomes_zero_for(*zero_params*)
Return True if the polynomial becomes zero if the parameters passed in *zero_params* are set to zero. Otherwise, return False.

Parameters **zero_params** – iterable of integers; The indices of the parameters to be checked.

copy()
Return a copy of a *Polynomial* or a subclass.

derive(*index*)
Generate the derivative by the parameter indexed *index*.

Parameters **index** – integer; The index of the paramater to derive by.

static from_expression(*expression*, *polysymbols*)
Alternative constructor. Construct the polynomial from an algebraic expression.

Parameters

- **expression** – string or sympy expression; The algebraic representation of the polynomial, e.g. "5*x1**2 + x1*x2"
- **polysymbols** – iterable of strings or sympy symbols; The symbols to be interpreted as the polynomial variables, e.g. "['x1','x2']".

has_constant_term()
Return True if the polynomial can be written as:

$$const + ...$$

Otherwise, return False.

replace(*expression*, *index*, *value*, *remove=False*)
Replace a variable in an expression by a number or a symbol. The entries in all `expolist` of the underlying *Polynomial* are set to zero. The coefficients are modified according to *value* and the powers indicated in the `expolist`.

Parameters

- **expression** – `_Expression`; The expression to replace the variable.
- **index** – integer; The index of the variable to be replaced.
- **value** – number or sympy expression; The value to insert for the chosen variable.
- **remove** – bool; Whether or not to remove the replaced parameter from the `parameters` in the *expression*.

simplify()
Combine terms that have the same exponents of the variables.

class `pySecDec.algebra.`**Pow**(*base*, *exponent*, *copy=True*)
Exponential. Store two expressions `A` and `B` to be interpreted as the exponential `A**B`.

Parameters

- **base** – `_Expression`; The base `A` of the exponential.
- **exponent** – `_Expression`; The exponent `B`.

- **copy** – bool; Whether or not to copy *base* and *exponent*.

copy()
Return a copy of a *Pow*.

derive(*index*)
Generate the derivative by the parameter indexed *index*.

Parameters **index** – integer; The index of the paramater to derive by.

replace(*expression*, *index*, *value*, *remove=False*)
Replace a variable in an expression by a number or a symbol. The entries in all `expolist` of the underlying *Polynomial* are set to zero. The coefficients are modified according to *value* and the powers indicated in the `expolist`.

Parameters

- **expression** – `_Expression`; The expression to replace the variable.
- **index** – integer; The index of the variable to be replaced.
- **value** – number or sympy expression; The value to insert for the chosen variable.
- **remove** – bool; Whether or not to remove the replaced parameter from the `parameters` in the *expression*.

simplify()
Apply the identity <something>**0 = 1 or <something>**1 = something if possible. Convert to *ExponentiatedPolynomial* if possible.

class `pySecDec.algebra.`**Product**(**factors*, ***kwargs*)
Product of polynomials. Store one or polynomials p_i to be interpreted as product $\prod_i p_i$.

Parameters

- **factors** – arbitrarily many instances of *Polynomial*; The factors p_i.
- **copy** – bool; Whether or not to copy the *factors*.

p_i can be accessed with `self.factors[i]`.

Example:

```
p = Product(p0, p1)
p0 = p.factors[0]
p1 = p.factors[1]
```

copy()
Return a copy of a *Product*.

derive(*index*)
Generate the derivative by the parameter indexed *index*. Return an instance of the optimized *ProductRule*.

Parameters **index** – integer; The index of the paramater to derive by.

replace(*expression*, *index*, *value*, *remove=False*)
Replace a variable in an expression by a number or a symbol. The entries in all `expolist` of the underlying *Polynomial* are set to zero. The coefficients are modified according to *value* and the powers indicated in the `expolist`.

Parameters

- **expression** – `_Expression`; The expression to replace the variable.

- **index** – integer; The index of the variable to be replaced.
- **value** – number or sympy expression; The value to insert for the chosen variable.
- **remove** – bool; Whether or not to remove the replaced parameter from the `parameters` in the *expression*.

simplify()

If one or more of `self.factors` is a *Product*, replace it by its factors. If only one factor is present, return that factor. Remove factors of one and zero.

class pySecDec.algebra.**ProductRule**(**expressions*, ***kwargs*)

Store an expression of the form

$$\sum_i c_i \prod_j \prod_k \left(\frac{d}{dx_k} \right)^{n_{ijk}} f_j \left(\{x_k\} \right)$$

The main reason for introducing this class is a speedup when calculating derivatives. In particular, this class implements simplifications such that the number of terms grows less than exponentially (scaling of the naive implementation of the product rule) with the number of derivatives.

Parameters expressions – arbitrarily many expressions; The expressions f_j.

copy()

Return a copy of a *ProductRule*.

derive(*index*)

Generate the derivative by the parameter indexed *index*. Note that this class is particularly designed to hold derivatives of a product.

Parameters index – integer; The index of the paramater to derive by.

replace(*index*, *value*, *remove=False*)

Replace a variable in an expression by a number or a symbol. The entries in all `expolist` of the underlying *Polynomial* are set to zero. The coefficients are modified according to *value* and the powers indicated in the `expolist`.

Parameters

- **expression** – `_Expression`; The expression to replace the variable.
- **index** – integer; The index of the variable to be replaced.
- **value** – number or sympy expression; The value to insert for the chosen variable.
- **remove** – bool; Whether or not to remove the replaced parameter from the `parameters` in the *expression*.

simplify()

Combine terms that have the same derivatives of the *expressions*.

class pySecDec.algebra.**Sum**(**summands*, ***kwargs*)

Sum of polynomials. Store one or polynomials p_i to be interpreted as product $\sum_i p_i$.

Parameters

- **summands** – arbitrarily many instances of *Polynomial*; The summands p_i.
- **copy** – bool; Whether or not to copy the *summands*.

p_i can be accessed with `self.summands[i]`.

Example:

```
p = Sum(p0, p1)
p0 = p.summands[0]
p1 = p.summands[1]
```

copy()

Return a copy of a *Sum*.

derive(*index*)

Generate the derivative by the parameter indexed *index*.

Parameters **index** – integer; The index of the paramater to derive by.

replace(*expression*, *index*, *value*, *remove=False*)

Replace a variable in an expression by a number or a symbol. The entries in all `expolist` of the underlying *Polynomial* are set to zero. The coefficients are modified according to *value* and the powers indicated in the `expolist`.

Parameters

- **expression** – `_Expression`; The expression to replace the variable.
- **index** – integer; The index of the variable to be replaced.
- **value** – number or sympy expression; The value to insert for the chosen variable.
- **remove** – bool; Whether or not to remove the replaced parameter from the `parameters` in the *expression*.

simplify()

If one or more of `self.summands` is a *Sum*, replace it by its summands. If only one summand is present, return that summand. Remove zero from sums.

class `pySecDec.decomposition.`**Sector**(*cast*, *other=[]*, *Jacobian=None*)

Container class for sectors that arise during the sector decomposition.

Parameters

- **cast** – iterable of *algebra.Product* or of *algebra.Polynomial*; The polynomials to be cast to the form *<monomial> * (const + ...)*
- **other** – iterable of *algebra.Polynomial*, optional; All variable transformations are applied to these polynomials but it is not attempted to achieve the form *<monomial> * (const + ...)*
- **Jacobian** – *algebra.Polynomial* with one term, optional; The Jacobian determinant of this sector. If not provided, the according unit monomial (1*x0^0*x1^0...) is assumed.

`pySecDec.decomposition.`**hide**(*polynomial*, *count*)

Hide the last *count* variables of a polynomial. This function is meant to be used before instantiating a *Sector*. It splits the `expolist` and the `polysymbols` at the index `count`.

See also:

unhide()

> **Warning:** The *polynomial* is **NOT** copied.

`pySecDec.decomposition.`**unhide**(*polynomial1*, *polynomial2*)

Undo the operation *hide()*; i.e. `unhide(*hide(polynomial))` is equal to `polynomial`.

See also:

hide()

Warning: *polynomial1* is modified in place.

The iterative sector decomposition routines

exception pySecDec.decomposition.iterative.**EndOfDecomposition**

This exception is raised if the function *iteration_step()* is called although the sector is already in standard form.

pySecDec.decomposition.iterative.**iteration_step**(*sector*)

Run a single step of the iterative sector decomposition as described in chapter 3.2 (part II) of arXiv:0803.4177v2 *[Hei08]*. Return an iterator of *Sector* - the arising subsectors.

Parameters **sector** – *Sector*; The sector to be decomposed.

pySecDec.decomposition.iterative.**iterative_decomposition**(*sector*)

Run the iterative sector decomposition as described in chapter 3.2 (part II) of arXiv:0803.4177v2 *[Hei08]*. Return an iterator of *Sector* - the arising subsectors.

Parameters **sector** – *Sector*; The sector to be decomposed.

pySecDec.decomposition.iterative.**primary_decomposition**(*sector*)

Perform the primary decomposition as described in chapter 3.2 (part I) of arXiv:0803.4177v2 *[Hei08]*. Return a list of *Sector* - the primary sectors. For *N* Feynman parameters, there are *N* primary sectors where the *i*-th Feynman parameter is set to *1* in sector *i*.

See also:

primary_decomposition_polynomial()

Parameters **sector** – *Sector*; The container holding the polynomials (typically U and F) to eliminate the Dirac delta from.

pySecDec.decomposition.iterative.**primary_decomposition_polynomial**(*polynomial*)

Perform the primary decomposition on a single polynomial.

See also:

primary_decomposition()

Parameters **polynomial** – *algebra.Polynomial*; The polynomial to eliminate the Dirac delta from.

pySecDec.decomposition.iterative.**remap_parameters**(*singular_parameters*, *Jacobian*, **polynomials*)

Remap the Feynman parameters according to eq. (16) of arXiv:0803.4177v2 *[Hei08]*. The parameter whose index comes first in *singular_parameters* is kept fix.

The remapping is done in place; i.e. the *polynomials* are **NOT** copied.

Parameters

- **singular_parameters** – list of integers; The indices α_r such that at least one of *polynomials* becomes zero if all $t_{\alpha_r} \to 0$.
- **Jacobian** – *Polynomial* with one term and no coefficients; The Jacobian determinant is multiplied to this polynomial.

- **polynomials** – abritrarily many instances of *algebra.Polynomial* where all of these have an equal number of variables; The polynomials of Feynman parameters to be remapped. These are typically F and U.

Example:

```
remap_parameters([1,2], Jacobian, F, U)
```

The geometric sector decomposition routines

`pySecDec.decomposition.geometric.`**`Cheng_Wu`**(*sector*, *index=-1*)

Replace one Feynman parameter by one. This means integrating out the Dirac delta according to the Cheng-Wu theorem.

Parameters

- **sector** – *Sector*; The container holding the polynomials (typically U and F) to eliminate the Dirac delta from.
- **index** – integer, optional; The index of the Feynman parameter to eliminate. Default: -1 (the last Feynman parameter)

class `pySecDec.decomposition.geometric.`**`Polytope`**(*vertices=None*, *facets=None*)

Representation of a polytope defined by either its vertices or its facets. Call *complete_representation()* to translate from one to the other representation.

Parameters

- **vertices** – two dimensional array; The polytope in vertex representation. Each row is interpreted as one vertex.
- **facets** – two dimensional array; The polytope in facet representation. Each row represents one facet F. A row in *facets* is interpreted as one normal vector n_F with additionally the constant a_F in the last column. The points v of the polytope obey

$$\bigcap_F (\langle n_F, v\rangle + a_F) \geq 0$$

`complete_representation`(*normaliz='normaliz'*, *workdir='normaliz_tmp'*, *keep_workdir=False*)

Transform the vertex representation of a polytope to the facet representation or the other way round. Remove surplus entries in `self.facets` or `self.vertices`.

Note: This function calls the command line executable of normaliz *[BIR]*. It is designed for *normaliz* version 3.0.0

Parameters

- **normaliz** – string; The shell command to run *normaliz*.
- **workdir** – string; The directory for the communication with *normaliz*. A directory with the specified name will be created in the current working directory. If the specified directory name already exists, an `OSError` is raised.

  Note: The communication with *normaliz* is done via files.

- **keep_workdir** – bool; Whether or not to delete the *workdir* after execution.

vertex_incidence_lists()

Return for each vertex the list of facets it lies in (as dictonary). The keys of the output dictonary are the vertices while the values are the indices of the facets in `self.facets`.

`pySecDec.decomposition.geometric.`**convex_hull**(**polynomials*)

Calculate the convex hull of the Minkowski sum of all polynomials in the input. The algorithm sets all coefficients to one first and then only keeps terms of the polynomial product that have coefficient 1. Return the list of these entries in the expolist of the product of all input polynomials.

Parameters polynomials – abritrarily many instances of `Polynomial` where all of these have an equal number of variables; The polynomials to calculate the convex hull for.

`pySecDec.decomposition.geometric.`**geometric_decomposition**(*sector*, *normaliz='normaliz'*, *workdir='normaliz_tmp'*)

Run the sector decomposition using the geomethod as described in *[BHJ+15]*.

Parameters

- **sector** – `Sector`; The sector to be decomposed.
- **normaliz** – string; The shell command to run *normaliz*.
- **workdir** – string; The directory for the communication with *normaliz*. A directory with the specified name will be created in the current working directory. If the specified directory name already exists, an `OSError` is raised.

Note: The communication with *normaliz* is done via files.

`pySecDec.decomposition.geometric.`**transform_variables**(*polynomial*, *transformation*, *polysymbols='y'*)

Transform the parameters x_i of a `pySecDec.algebra.Polynomial`,

$$x_i \rightarrow \prod_j x_j^{T_{ij}}$$

, where T_{ij} is the transformation matrix.

Parameters

- **polynomial** – `pySecDec.algebra.Polynomial`; The polynomial to transform the variables in.
- **transformation** – two dimensional array; The transformation matrix T_{ij}.
- **polysymbols** – string or iterable of strings; The symbols for the new variables. This argument is passed to the default constructor of `pySecDec.algebra.Polynomial`. Refer to the documentation of `pySecDec.algebra.Polynomial` for further details.

`pySecDec.decomposition.geometric.`**triangulate**(*cone*, *normaliz='normaliz'*, *workdir='normaliz_tmp'*, *keep_workdir=False*)

Split a cone into simplicial cones; i.e. cones defined by exactly D rays where D is the dimensionality.

Note: This function calls the command line executable of normaliz. It is designed for *normaliz* version 3.0.0

Parameters

- **cone** – two dimensional array; The defining rays of the cone.

- **normaliz** – string; The shell command to run *normaliz*.
- **workdir** – string; The directory for the communication with *normaliz*. A directory with the specified name will be created in the current working directory. If the specified directory name already exists, an `OSError` is raised.

  Note: The communication with *normaliz* is done via files.

- **keep_workdir** – bool; Whether or not to delete the *workdir* after execution.

miscellaneous routines

`pySecDec.misc.`**`adjugate`**`(`*M*`)`

Calculate the adjugate of a matrix.

Parameters **M** – a square-matrix-like array;

`pySecDec.misc.`**`all_pairs`**`(`*iterable*`)`

Return all possible pairs of a given set. `all_pairs([1,2,3,4]) --> [(1,2),(3,4)] [(1,3),(2,4)] [(1,4),(2,3)]`

Parameters **iterable** – iterable; The set to be split into all possible pairs.

`pySecDec.misc.`**`argsort_2D_array`**`(`*array*`)`

Sort a 2D array according to its row entries. The idea is to bring identical rows together.

See also:

If your array is not two dimesional use *argsort_ND_array()*.

Example:

input		sorted
1 2 3		1 2 3
2 3 4		1 2 3
1 2 3		2 3 4

Return the indices like numpy's `argsort()` would.

Parameters **array** – 2D array; The array to be argsorted.

`pySecDec.misc.`**`argsort_ND_array`**`(`*array*`)`

Like *argsort_2D_array()*, this function groups identical entries in an array with any dimensionality greater than (or equal to) two together.

Return the indices like numpy's `argsort()` would.

See also:

argsort_2D_array()

Parameters **array** – ND array, $N >= 2$; The array to be argsorted.

`pySecDec.misc.`**`assert_degree_at_most_max_degree`**`(`*expression*, *variables*, *max_degree*, *error_message*`)`

Assert that *expression* is a polynomial of degree less or equal *max_degree* in the *variables*.

`pySecDec.misc.`**`cached_property`**`(`*method*`)`

Like the builtin *property* to be used as decorator but the method is only called once per instance.

Example:

```
class C(object):
    'Sum up the numbers from one to `N`.'
    def __init__(self, N):
        self.N = N
    @cached_property
    def sum(self):
        result = 0
        for i in range(1, self.N + 1):
            result += i
        return result
```

pySecDec.misc.**det** (*M*)

Calculate the determinant of a matrix.

Parameters **M** – a square-matrix-like array;

pySecDec.misc.**doc** (*docstring*)

Decorator that replaces a function's docstring with *docstring*.

Example:

```
@doc('documentation of `some_funcion`')
def some_function(*args, **kwargs):
    pass
```

pySecDec.misc.**missing** (*full*, *part*)

Return the elements in *full* that are not contained in *part*. Raise *ValueError* if an element is in *part* but not in *full*. `missing([1,2,3],[1]) --> [2,3]` `missing([1,2,3,1],[1,2]) --> [3,1]` `missing([1,2,3],[1,'a']) --> ValueError`

Parameters

- **full** – iterable; The set of elements to complete *part* with.
- **part** – iterable; The set to be completed to a superset of *full*.

pySecDec.misc.**powerset** (*iterable*, *exclude_empty=False*, *stride=1*)

Return an iterator over the powerset of a given set. `powerset([1,2,3]) --> () (1,) (2,) (3,) (1,2) (1,3) (2,3) (1,2,3)`

Parameters

- **iterable** – iterable; The set to generate the powerset for.
- **exclude_empty** – bool, optional; If True, skip the empty set in the powerset. Default is False.
- **stride** – integer; Only generate sets that have a multiple of *stride* elements. `powerset([1,2,3],stride=2) --> () (1,2) (1,3) (2,3)`

pySecDec.misc.**sympify_symbols** (*iterable*, *error_message*, *allow_number=False*)

sympify each item in *iterable* and assert that it is a *symbol*.

Routines to Feynman parametrize a loop integral

class pySecDec.loop_integral.**LoopIntegral** (**args*, ***kwargs*)

Container class for loop integrals. The main purpose of this class is to convert a loop integral from the momentum representation to the Feynman parameter representation.

It is possible to provide either the graph of the loop integrals as adjacency list, or the propagators.

The Feynman parametrized integral is a product of the following expressions that are accessible as member properties:

- `self.regulator ** self.regulator_power`
- `self.Gamma_factor`
- `self.exponentiated_U`
- `self.exponentiated_F`
- `self.numerator`
- `self.measure`,

where `self` is an instance of either *LoopIntegralFromGraph* or *LoopIntegralFromPropagators*.

Whereas `self.numerator` describes the numerator polynomial generated by tensor numerators or inverse propagators, `self.measure` contains the monomial associated with the integration measure in the case of propagator powers $\neq 1$. The Gamma functions in the denominator belonging to the measure, however, are multiplied to the overall Gamma factor given by `self.Gamma_factor`. The overall sign $(-1)^{N_\nu}$ is included in `self.numerator`.

See also:

- input as graph: *LoopIntegralFromGraph*
- input as list of propagators: *LoopIntegralFromPropagators*

class `pySecDec.loop_integral.`**`LoopIntegralFromGraph`**(*internal_lines, external_lines, replacement_rules=[], Feynman_parameters='x', regulator='eps', regulator_power=0, dimensionality='4-2*eps', powerlist=[]*)

Construct the Feynman parametrization of a loop integral from the graph using the cut construction method.

Example:

```
>>> from pySecDec.loop_integral import *
>>> internal_lines = [['0',[1,2]], ['m',[2,3]], ['m',[3,1]]]
>>> external_lines = [['p1',1],['p2',2],['-p12',3]]
>>> li = LoopIntegralFromGraph(internal_lines, external_lines)
>>> li.exponentiated_U
( + (1)*x0 + (1)*x1 + (1)*x2)**(2*eps - 1)
>>> li.exponentiated_F
( + (m**2)*x2**2 + (2*m**2 - p12**2)*x1*x2 + (m**2)*x1**2 + (m**2 - p1**2)*x0*x2␣
↪+ (m**2 - p2**2)*x0*x1)**(-eps - 1)
```

Parameters

- **`internal_lines`** – iterable of internal line specification, consisting of string or sympy expression for mass and a pair of strings or numbers for the vertices, e.g. [['m', [1,2]], ['0', [2,1]]].
- **`external_lines`** – iterable of external line specification, consisting of string or sympy expression for external momentum and a strings or number for the vertex, e.g. [['p1', 1], ['p2', 2]].
- **`replacement_rules`** – iterable of iterables with two strings or sympy expressions, optional; Symbolic replacements to be made for the external momenta, e.g. definition of Mandelstam variables. Example: [('p1*p2', 's'), ('p1**2', 0)] where `p1` and `p2` are external momenta. It is also possible to specify vector replacements, for example [('p4', '-(p1+p2+p3)')].

- **Feynman_parameters** – iterable or string, optional; The symbols to be used for the Feynman parameters. If a string is passed, the Feynman parameter variables will be consecutively numbered starting from zero.
- **regulator** – string or sympy symbol, optional; The symbol to be used for the dimensional regulator (typically ϵ or ϵ_D)

  Note: If you change this symbol, you have to adapt the *dimensionality* accordingly.

- **regulator_power** – integer; The regulator to this power will be multiplied by the numerator.
- **dimensionality** – string or sympy expression, optional; The dimensionality; typically $4 - 2\epsilon$, which is the default value.
- **powerlist** – iterable, optional; The powers of the propergators, possibly dependent on the *regulator*. In case of negative powers, the derivative with respect to the corresponding Feynman parameter is calculated as explained in Section 3.2.4 of Ref. *[BHJ+15]*. If negative powers are combined with a tensor numerator, the derivative acts on the Feynman-parametrized tensor numerator as well, which should lead to a consistent result.

class pySecDec.loop_integral.**LoopIntegralFromPropagators**(*propagators*, *loop_momenta*, *external_momenta=[]*, *Lorentz_indices=[]*, *numerator=1*, *metric_tensor='g'*, *replacement_rules=[]*, *Feynman_parameters='x'*, *regulator='eps'*, *regulator_power=0*, *dimensionality='4-2*eps'*, *powerlist=[]*)

Construct the Feynman parametrization of a loop integral from the algebraic momentum representation.

See also:

[Hei08], *[GKR+11]*

Example:

```
>>> from pySecDec.loop_integral import *
>>> propagators = ['k**2', '(k - p)**2']
>>> loop_momenta = ['k']
>>> li = LoopIntegralFromPropagators(propagators, loop_momenta)
>>> li.exponentiated_U
( + (1)*x0 + (1)*x1)**(2*eps - 2)
>>> li.exponentiated_F
( + (-p**2)*x0*x1)**(-eps)
```

The 1st (U) and 2nd (F) Symanzik polynomials and their exponents can also be accessed independently:

```
>>> li.U
 + (1)*x0 + (1)*x1
>>> li.F
 + (-p**2)*x0*x1
>>>
>>> li.exponent_U
2*eps - 2
```

```
>>> li.exponent_F
-eps
```

Parameters

- **propagators** – iterable of strings or sympy expressions; The propagators, e.g. ['k1**2', '(k1-k2)**2 - m1**2'].
- **loop_momenta** – iterable of strings or sympy expressions; The loop momenta, e.g. ['k1','k2'].
- **external_momenta** – iterable of strings or sympy expressions, optional; The external momenta, e.g. ['p1','p2']. Specifying the *external_momenta* is only required when a *numerator* is to be constructed.
- **Lorentz_indices** – iterable of strings or sympy expressions, optional; Symbols to be used as Lorentz indices in the numerator.

See also:

parameter *numerator*

Parameters numerator – string or sympy expression, optional; The numerator of the loop integral. Scalar products must be passed in index notation e.g. "k1(mu)*k2(mu)". The numerator should be a sum of products of exclusively: * numbers * scalar products (e.g. "p1(mu)*k1(mu)*p1(nu)*k2(nu)") * *symbols* (e.g. "m")

Examples:

- 'p1(mu)*k1(mu)*p1(nu)*k2(nu) + 4*s*eps*k1(mu)*k1(mu)'
- 'p1(mu)*(k1(mu) + k2(mu))*p1(nu)*k2(nu)'
- 'p1(mu)*k1(mu)*my_function(eps)'

Hint: It is possible to use numbers as indices, for example 'p1(mu)*p2(mu)*k1(nu)*k2(nu) = p1(1)*p2(1)*k1(2)*k2(2)'.

Hint: The numerator may have uncontracted indices, e.g. 'k1(mu)*k2(nu)'

Warning: All Lorentz indices (including the contracted ones) must be explicitly defined using the parameter *Lorentz_indices*.

Warning: It is assumed that the numerator is and all its derivatives by the *regulator* are finite and defined if $\epsilon = 0$ is inserted explicitly. In particular, if user defined functions (like in the example `p1(mu)*k1(mu)*my_function(eps)`) appear, make sure that `my_function(0)` is finite.

Hint: In order to mimic a singular user defined function, use the parameter *regulator_power*. For example, instead of `numerator = gamma(eps)` you could enter `numerator = eps_times_gamma(eps)` in conjunction with `regulator_power = -1`

Warning: The *numerator* is very flexible in its input. However, that flexibility comes for the price of less error safety. We have no way of checking for all possible mistakes in the input. If your numerator is more advanced than in the examples above, you should proceed with great caution.

Parameters

- **metric_tensor** – string or sympy symbol, optional; The symbol to be used for the (Minkowski) metric tensor $g^{\mu\nu}$.
- **replacement_rules** – iterable of iterables with two strings or sympy expressions, optional; Symbolic replacements to be made for the external momenta, e.g. definition of Mandelstam variables. Example: [('p1*p2', 's'), ('p1**2', 0)] where `p1` and `p2` are external momenta. It is also possible to specify vector replacements, for example [('p4', '-(p1+p2+p3)')].
- **Feynman_parameters** – iterable or string, optional; The symbols to be used for the Feynman parameters. If a string is passed, the Feynman parameter variables will be consecutively numbered starting from zero.
- **regulator** – string or sympy symbol, optional; The symbol to be used for the dimensional regulator (typically ϵ or ϵ_D)

  Note: If you change this symbol, you have to adapt the *dimensionality* accordingly.

- **regulator_power** – integer; The regulator to this power will be multiplied by the numerator.
- **dimensionality** – string or sympy expression, optional; The dimensionality; typically $4-2\epsilon$, which is the default value.
- **powerlist** – iterable, optional; The powers of the propergators, possibly dependent on the *regulator*. In case of negative powers, the derivative with respect to the corresponding Feynman parameter is calculated as explained in Section 3.2.4 of Ref. *[BHJ+15]*. If negative powers are combined with a tensor numerator, the derivative acts on the Feynman-parametrized tensor numerator as well, which should lead to a consistent result.

Routines to isolate the divergencies in an ϵ expansion

`pySecDec.subtraction.`**`integrate_pole_part`**(*polyprod*, **indices*)

Transform an integral of the form

$$\int_0^1 dt_j t_j^{(a-b\epsilon_1-c\epsilon_2+...)} \mathcal{I}(\sqcup_|, \{\sqcup_{\rangle\neq|}\}, \epsilon_\infty, \epsilon_\in, ...)$$

into the form

$$\sum_{p=0}^{|a|-1} \frac{1}{a+p+1-b\epsilon_1-c\epsilon_2-...} \frac{\mathcal{I}^{(\surd)}(\prime, \{\sqcup_{\rangle\neq|}\}, \epsilon_\infty, \epsilon_\in, ...)}{p!} + \int_0^1 dt_j t_j^{(a-b\epsilon_1-c\epsilon_2+...)} R(t_j, \{t_{i\neq j}\}, \epsilon_1, \epsilon_2, ...)$$

, where $\mathcal{I}^{(\surd)}$ is denotes the p-th derivative of $\mathcal{I}$ with respect to t_j. The equations above are to be understood schematically.

See also:

This function implements the transformation from equation (19) to (21) as described in arXiv:0803.4177v2 *[Hei08]*.

Parameters

- **polyprod** – `algebra.Product` of the form `<product of <monomial>**(a_j + ...)> * <regulator poles of cal_I> * <cal_I>`; The input product as decribed above. The <product of <monomial>**(a_j + ...)> should be a `pySecDec.algebra.Product` of <monomial>**(a_j + ...). as described below. The <monomial>**(a_j + ...) should be an `pySecDec.algebra.ExponentiatedPolynomial` with `exponent` being a `Polynomial` of the regulators $\epsilon_1, \epsilon_2, \ldots$. Although no dependence on the Feynman parameters is expected in the `exponent`, the polynomial variables should be the Feynman parameters and the regulators. The constant term of the exponent should be numerical. The polynomial variables of `monomial` and the other factors (interpreted as $\mathcal{I}$) are interpreted as the Feynman parameters and the epsilon regulators. Make sure that the last factor (`<cal_I>`) is defined and finite for $\epsilon = 0$. All poles for $\epsilon \rightarrow 0$ should be made explicit by putting them into `<regulator poles of cal_I>` as `pySecDec.algebra.Pow` with `exponent = -1` and the `base` of type `pySecDec.algebra.Polynomial`.
- **indices** – arbitrarily many integers; The index/indices of the parameter(s) to partially integrate. j in the formulae above.

Return the pole part and the numerically integrable remainder as a list. That is the sum and the integrand of equation (21) in arXiv:0803.4177v2 *[Hei08]*. Each returned list element has the same structure as the input *polyprod*.

Routines to series expand singular and nonsingular expressions

`pySecDec.expansion.`**`expand_Taylor`**`(`*expression*, *indices*, *orders*`)`

Series/Taylor expand a nonsingular *expression* around zero.

Return a `algebra.Polynomial` - the series expansion.

Parameters

- **expression** – an expression composed of the types defined in the module `algebra`; The expression to be series expanded.
- **indices** – integer or iterable of integers; The indices of the parameters to expand. The ordering of the indices defines the ordering of the expansion.
- **order** – integer or iterable of integers; The order to which the expansion is to be calculated.

`pySecDec.expansion.`**`expand_singular`**`(`*product*, *indices*, *orders*`)`

Series expand a potentially singular expression of the form

$$\frac{a_N \epsilon_0 + b_N \epsilon_1 + \ldots}{a_D \epsilon_0 + b_D \epsilon_1 + \ldots}$$

Return a `algebra.Polynomial` - the series expansion.

Parameters

- **product** – `algebra.Product` with factors of the form `<polynomial>` and `<polynomial> ** -1`; The expression to be series expanded.
- **indices** – integer or iterable of integers; The indices of the parameters to expand. The ordering of the indices defines the ordering of the expansion.
- **order** – integer or iterable of integers; The order to which the expansion is to be calculated.

CHAPTER

FOUR

REFERENCES

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

[BH00] T. Binoth and G. Heinrich *An automatized algorithm to compute infrared divergent multiloop integrals*, Nucl. Phys. B 585 (2000) 741,
doi:10.1016/S0550-3213(00)00429-6,
arXiv:hep-ph/0004013

[BHJ+15] S. Borowka, G. Heinrich, S. P. Jones, M. Kerner, J. Schlenk, T. Zirke *SecDec-3.0: numerical evaluation of multi-scale integrals beyond one loop*, 2015, Comput.Phys.Comm.196
doi:10.1016/j.cpc.2015.05.022,
arXiv:1502.06595

[BIR] W. Bruns and B. Ichim and T. Römer and R. Sieg and C. Söger *Normaliz. Algorithms for rational cones and affine monoids*,
available at https://www.normaliz.uni-osnabrueck.de

[GKR+11] J. Gluza, K. Kajda, T. Riemann, V. Yundin *Numerical Evaluation of Tensor Feynman Integrals in Euclidean Kinematics*, 2011, Eur.Phys.J.C71,
doi:10.1140/epjc/s10052-010-1516-y,
arXiv:1010.1667

[Hei08] G. Heinrich *Sector Decomposition*, 2008, Int.J.Mod.Phys.A23,
doi:10.1142/S0217751X08040263,
arXiv:0803.4177

PYTHON MODULE INDEX

M

P

R

S

T

U

V