pySecDec Documentation

Release 1.5.3

Sophia Borowka Gudrun Heinrich Stephan Jahn
Stephen Jones Matthias Kerner Florian Langer
Vitaly Magerya Andres Poldaru Johannes Schlenk
Emilio Villa Tom Zirke

Jan 26, 2022

CONTENTS

pySecDec [PSD17], [PSD18], [PSD21] is a toolbox for the calculation of dimensionally regulated parameter integrals using the sector decomposition approach [BH00]; see also [Hei08], [BHJ+15].

Please cite the following references if you use *pySecDec* for a scientific publication:

- pySecDec [PSD17], [PSD18], [PSD21]
- CUBA [Hah05], [Hah16]
- FORM [Ver00], [KUV13], [RUV17]
- GSL [GSL]
- nauty [MP+14] (if you use *dreadnaut*)
- normaliz [BIR], [BIS16] (if you use a geometric decomposition strategy)
- QMC [LWY+15] (if you use the quasi-monte carlo integrator)

CHAPTER

ONE

INSTALLATION

1.1 Download the Program and Install

pySecDec works under Python version 3.6 or newer on unix-like systems. The latest release can be installed from PyPI by first (optionally) upgrading pip:

```
$ python3 -m pip install --user 'pip>=20.1'
```

and then running:

```
$ python3 -m pip install --user --upgrade pySecDec
```

1.2 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 - currently versions `3.3.0`, `3.4.0`, `3.5.4`, `3.6.0`, `3.6.2`, `3.7.3`, `3.7.4`, and `3.8.1` are known to work. 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.

1.3 Drawing Feynman Diagrams with *neato*

In order to use *plot_diagram()*, the command line tool *neato* must be available. The function *loop_package()* tries to call *plot_diagram()* if given a *LoopIntegralFromGraph* and issues a warning on failure. That warning can be safely ignored if you are not interested in the drawing.

neato is part of the *graphviz* package. It is available in many package repositories and at http://www.graphviz.org.

1.4 Additional Dependencies for Generated c++ Packages

Note: The following packages are redistributed with the *pySecDec* tarball; i.e. you don't have to install any of them yourself.

The intended main usage of *pySecDec* is to make it write c++ packages using the functions `pySecDec.code_writer.make_package()` and `pySecDec.loop_integral.loop_package()`. In order to build these c++ packages, the following additional non-python-based libraries and programs are used:

- CUBA (http://www.feynarts.de/cuba/)
- QMC (https://github.com/mppmu/qmc)
- FORM (http://www.nikhef.nl/~form/)
- SecDecUtil (part of *pySecDec*, see *SedDecUtil*), depends on:
  - catch (https://github.com/philsquared/Catch)
  - gsl (http://www.gnu.org/software/gsl/)

The functions `pySecDec.code_writer.make_package()` and `pySecDec.loop_integral.loop_package()` can use the external program *nauty* [MP+14] to find all sector symmetries and therefore reduce the number of sectors:

- NAUTY (http://pallini.di.uniroma1.it/)

These packages are redistributed along with pySecDec itself, and will be built automatically during pySecDec installation.

CHAPTER

TWO

GETTING STARTED

After installation, you should have a folder *examples* in your main *pySecDec* directory. Here we describe a few of the examples available in the *examples* directory. A full list of examples is given in *List of Examples*.

2.1 A Simple Example

We first show how to compute a simple dimensionally regulated integral:

$$\int_0^1 \mathrm{d}x \int_0^1 \mathrm{d}y \, (x+y)^{-2+\epsilon}.$$

To run the example change to the *easy* directory and run the commands:

```
$ python3 generate_easy.py
$ make -C easy
$ python3 integrate_easy.py
```

Additional build options are discussed in the *next section*. This will evaluate and print the result of the integral:

```
Numerical Result: + ((1.0000000000000022e+00,0.0000000000000000e+00) +/- (5.
↪6535215397909540 1e-17,0.0000000000000000e+00))*eps^-1 + ((3.06852819440053548e-01,0.
↪0000000000000000e+00) +/- (1.18502493127591741e-15,0.0000000000000000e+00)) + O(eps)
Analytic Result: + (1.000000)*eps^-1 + (0.306853) + O(eps)
```

The file `generate_easy.py` defines the integral and calls *pySecDec* to perform the sector decomposition. When run it produces the directory *easy* which contains the code required to numerically evaluate the integral. The make command builds this code and produces a library. The file `integrate_easy.py` loads the integral library and evaluates the integral. The user is encouraged to copy and adapt these files to evaluate their own integrals.

Note: If the user is interested in evaluating a loop integral there are many convenience functions that make this much easier. Please see *Evaluating a Loop Integral* for more details.

In `generate_easy.py` we first import `make_package`, a function which can decompose, subtract and expand regulated integrals and write a C++ package to evaluate them. To define our integral we give it a *name* which will be used as the name of the output directory and C++ namespace. The *integration_variables* are declared along with a list of the name of the *regulators*. We must specify a list of the *requested_orders* to which *pySecDec* should expand our integral in each regulator. Here we specify `requested_orders = [0]` which instructs `make_package` to expand the integral up to and including $\mathcal{O}(\epsilon)$. Next, we declare the *polynomials_to_decompose*, here *sympy* syntax should be used.

```
#!/usr/bin/env python3

from pySecDec import make_package

if __name__ == "__main__":

    make_package(

    name = 'easy',
    integration_variables = ['x','y'],
    regulators = ['eps'],

    requested_orders = [0],
    polynomials_to_decompose = ['(x+y)^(-2+eps)'],
    )
```

Once the C++ library has been written and built we run `integrate_easy.py`. Here the library is loaded using *IntegralLibrary*. Calling the instance of *IntegralLibrary* with `easy_integral()` numerically evaluates the integral and returns the result.

```
#!/usr/bin/env python3

from pySecDec.integral_interface import IntegralLibrary
from math import log

if __name__ == "__main__":

    # load c++ library
    easy = IntegralLibrary('easy/easy_pylink.so')

    # integrate
    _, _, result = easy()

    # print result
    print('Numerical Result:' + result)
    print('Analytic Result:' + ' + (%f)*eps^-1 + (%f) + O(eps)' % (1.0,1.0-log(2.0)))
```

2.2 Evaluating a Loop Integral

A simple example of the evaluation of a loop integral with *pySecDec* is *box1L*. This example computes a one-loop box with one off-shell leg (with off-shellness `s1`) and one internal massive line (with mass squared `msq`), it is shown in Fig. 2.1.

To run the example change to the *box1L* directory and run the commands:

```
$ python3 generate_box1L.py
$ make -C box1L
$ python3 integrate_box1L.py
```

This will print the result of the integral evaluated with Mandelstam invariants `s=4.0`, `t=-0.75` and `s1=1.25`, `msq=1.0`:

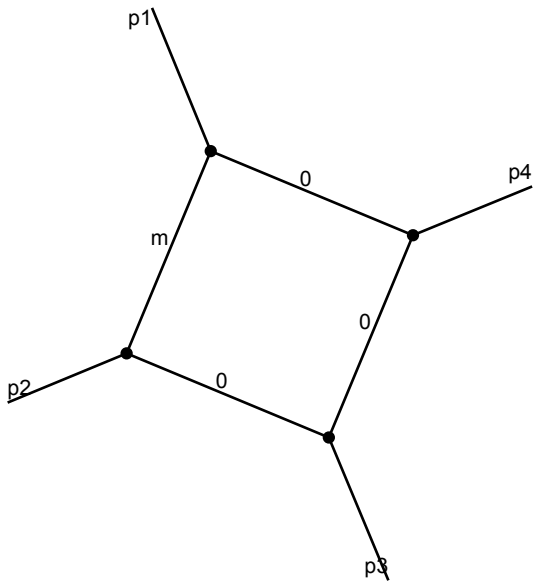

Fig. 2.1: Diagrammatic representation of *box1L*

```
eps^-2: -0.142868356275422825 - 1.63596224151119965e-6*I +/- ( 0.00118022544307414272 +␣
↪0.000210769456586696187*I )
eps^-1: 0.639405625715768089 + 1.3427703668990280 2e-6*I +/- ( 0.00650722394065588166 + 0.
↪000971496627153705891*I )
eps^0 : -0.425514350373418893 + 1.86892487760861536*I +/- ( 0.00706834403694714484 + 0.
↪0186497890361357298*I )
```

The file `generate_box1L.py` defines the loop integral and calls *pySecDec* to perform the sector decomposition. When run it produces the directory *box1L* which contains the code required to numerically evaluate the integral. The make command builds this code and produces a library. The file `integrate_box1L.py` loads the integral library and evaluates the integral for a specified numerical point.

The content of the python files is described in detail in the following sections. The user is encouraged to copy and adapt these files to evaluate their own loop integrals.

2.2.1 Defining a Loop Integral

To explain the input format, let us look at `generate_box1L.py` from the one-loop box example. The first two lines read

```
from pySecDec.loop_integral import loop_package
import pySecDec as psd
```

They say that the module *pySecDec* should be imported with the alias *psd*, and that the function *loop_package* from the module *loop_integral* is needed.

The following part contains the definition of the loop integral `li`:

```
li = psd.loop_integral.LoopIntegralFromGraph(
# give adjacency list and indicate whether the propagator connecting the numbered␣
↪vertices is massive or massless in the first entry of each list item.
internal_lines = [['m',[1,2]],[0,[2,3]],[0,[3,4]],[0,[4,1]]],
# contains the names of the external momenta and the label of the vertex they are␣
↪attached to
external_lines = [['p1',1],['p2',2],['p3',3],['p4',4]],

# define the kinematics and the names for the kinematic invariants
replacement_rules = [
                        ('p1*p1', 's1'),
                        ('p2*p2', 0),
                        ('p3*p3', 0),
                        ('p4*p4', 0),
                        ('p3*p2', 't/2'),
                        ('p1*p2', 's/2-s1/2'),
                        ('p1*p4', 't/2-s1/2'),
                        ('p2*p4', 's1/2-t/2-s/2'),
                        ('p3*p4', 's/2'),
                        ('m**2', 'msq')
                    ]
)
```

Here the class *LoopIntegralFromGraph* is used to Feynman parametrize the loop integral given the adjacency list. Alternatively, the class *LoopIntegralFromPropagators* can be used to construct the Feynman integral given the momentum representation.

The symbols for the kinematic invariants and the masses also need to be given as an ordered list. The ordering is important as the numerical values assigned to these list elements at the numerical evaluation stage should have the same order.

```
Mandelstam_symbols = ['s','t','s1']
mass_symbols = ['msq']
```

Next, the function *loop_package* is called. It will create a folder called *box1L*. It performs the algebraic sector decomposition steps and writes a package containing the C++ code for the numerical evaluation. The argument *requested_orders* specifies the order in the regulator to which the integral should be expanded. For a complete list of possible options see *loop_package*.

```
loop_package(

name = 'box1L',

loop_integral = li,

real_parameters = Mandelstam_symbols + mass_symbols,

# the highest order of the final epsilon expansion --> change this value to whatever you␣
↪think is appropriate
requested_orders = [0],

# the optimization level to use in FORM (can be 0, 1, 2, 3, 4)
form_optimization_level = 2,

# the WorkSpace parameter for FORM
form_work_space = '100M',

# the method to be used for the sector decomposition
# valid values are ``iterative`` or ``geometric`` or ``geometric_ku``
decomposition_method = 'iterative',
# if you choose ``geometric[_ku]`` and 'normaliz' is not in your
# $PATH, you can set the path to the 'normaliz' command-line
# executable here
#normaliz_executable='/path/to/normaliz',

)
```

2.2.2 Building the C++ Library

After running the python script *generate_box1L.py* the folder *box1L* is created and should contain the following files and subdirectories

```
Makefile    README    box1L.pdf    box1L_integral    integral_names.txt    pylink
Makefile.conf    box1L.hpp    box1L_coefficients    integrate_box1L.cpp    src
```

in the folder *box1L*, typing

```
$ make
```

will create the static library `box1L_integral/libbox1L_integral.a` and `box1L_pylink.so` which can be linked

to external programs. The `make` command can also be run in parallel by using the `-j` option. The number of threads each instance of `tform` uses can be set via the environment variable *FORMTHREADS*.

New in version 1.4: The environment variable *FORMOPT* sets FORM's code optimization level. If not set, the value that was passed to *make_package* or *loop_package* is used.

To build the dynamic library `libbox1L.so` set `dynamic` as build target:

```
$ make dynamic
```

To build the library with *nvcc* for GPU support, type

```
$ CXX=nvcc SECDEC_WITH_CUDA_FLAGS="-arch=sm_XX" make
```

where `sm_XX` must be replaced by the target GPU architechtures, see the arch option of NVCC. The `SECDEC_WITH_CUDA_FLAGS` environment variable, which enables GPU code compilation, contains flags which are passed to NVCC during code compilation and linking. Multiple GPU architectures may be specified as described in the NVCC manual, for example `SECDEC_WITH_CUDA_FLAGS="-gencode arch=compute_XX,code=sm_XX -gencode arch=compute_YY,code=sm_YY"` where `XX` and `YY` are the target GPU architectures. The script `examples/easy/print-cuda-arch.sh` can be used to obtain the compute architecture of your current machine.

To evaluate the integral numerically a program can call one of these libraries. How to do this interactively or via a python script is explained in the section *Python Interface*. Alternatively, a C++ program can be produced as explained in the section *C++ Interface*.

2.2.3 Python Interface (basic)

To evaluate the integral for a given numerical point we can use `integrate_box1L.py`. First it imports the necessary python packages and loads the C++ library.

```
from __future__ import print_function
from pySecDec.integral_interface import IntegralLibrary
import sympy as sp

# load c++ library
box1L = IntegralLibrary('box1L/box1L_pylink.so')
```

Next, an integrator is configured for the numerical integration. The full list of available integrators and their options is given in *integral_interface*.

```
# choose integrator
box1L.use_Vegas(flags=2) # ``flags=2``: verbose --> see Cuba manual
```

If you want to use GPUs, change to the *CudaQmc* integrator. For example, to run on all available GPUs and CPU cores using the Korobov transform with weight 3, change the above lines to

```
# choose integrator
box1L.use_Qmc(transform='Korobov3')
```

Calling the `box` library numerically evaluates the integral. Note that the order of the real parameters must match that specified in `generate_box1L.py`. A list of possible settings for the library, in particular details of how to set the contour deformation parameters, is given in *IntegralLibrary*. To change the accuracy settings of the integration, the most important parameters are `epsrel`, `epsabs` and `maxeval`, which can be added to the integrator argument list:

```
# choose integrator
box1L.use_Vegas(flags=2,epsrel=0.01, epsabs=1e-07, maxeval=1000000)
```

```
# integrate
str_integral_without_prefactor, str_prefactor, str_integral_with_prefactor = box1L(real_
↪parameters=[4.0, -0.75, 1.25, 1.0])
```

In case of a sign check error (sign_check_error), the arguments `number_of_presamples`, `deformation_parameters_maximum`, and `deformation_parameters_minimum` as described in *IntegralLibrary* can be used to modify the contour. At this point the string `str_integral_with_prefactor` contains the full result of the integral and can be manipulated as required. In the `integrate_box1L.py` an example is shown how to parse the expression with *sympy* and access individual orders of the regulator.

Note: Instead of parsing the result, it can simply be printed with the line `print(str_integral_with_prefactor)`.

```
# convert complex numbers from c++ to sympy notation
str_integral_with_prefactor = str_integral_with_prefactor.replace(',','+I*')
str_prefactor = str_prefactor.replace(',','+I*')
str_integral_without_prefactor = str_integral_without_prefactor.replace(',','+I*')

# convert result to sympy expressions
integral_with_prefactor = sp.sympify(str_integral_with_prefactor.replace('+/-',
↪'*value+error*'))
integral_with_prefactor_err = sp.sympify(str_integral_with_prefactor.replace('+/-',
↪'*value+error*'))
prefactor = sp.sympify(str_prefactor)
integral_without_prefactor = sp.sympify(str_integral_without_prefactor.replace('+/-',
↪'*value+error*'))
integral_without_prefactor_err = sp.sympify(str_integral_without_prefactor.replace('+/-',
↪'*value+error*'))

# examples how to access individual orders
print('Numerical Result')
print('eps^-2:', integral_with_prefactor.coeff('eps',-2).coeff('value'), '+/- (',␣
↪integral_with_prefactor_err.coeff('eps',-2).coeff('error'), ')')
print('eps^-1:', integral_with_prefactor.coeff('eps',-1).coeff('value'), '+/- (',␣
↪integral_with_prefactor_err.coeff('eps',-1).coeff('error'), ')')
print('eps^0 :', integral_with_prefactor.coeff('eps',0).coeff('value'), '+/- (',␣
↪integral_with_prefactor_err.coeff('eps',0).coeff('error'), ')')
```

An example of how to loop over several kinematic points is shown in the example *integrate_box1L_multiple_points.py*.

2.2.4 C++ Interface (advanced)

Usually it is easier to obtain a numerical result using the *Python Interface*. However, the library can also be used directly from C++. Inside the generated *box1L* folder the file `integrate_box1L.cpp` demonstrates this.

After the lines parsing the input parameters, an *secdecutil::Integrator* is constructed and its parameters are set:

```
// Set up Integrator
secdecutil::integrators::Qmc<
                            box1L::integrand_return_t,
                                box1L::maximal_number_of_integration_variables,
                                integrators::transforms::Korobov<3>::type,
```

(continues on next page)

(continued from previous page)

```
                                box1L::user_integrand_t
                            > integrator;
integrator.verbosity = 1;
```

The amplitude is constructed via a call to *name::make_amplitudes()* and packed into a `name::handler_t`.

```
// Construct the amplitudes
std::vector<box1L::nested_series_t<box1L::sum_t>> unwrapped_amplitudes =
    box1L::make_amplitudes(real_parameters, complex_parameters, "box1L_coefficients",␣
↪integrator);

// Pack amplitudes into handler
box1L::handler_t<box1L::amplitudes_t> amplitudes
(
    unwrapped_amplitudes,
    integrator.epsrel, integrator.epsabs
    // further optional arguments: maxeval, mineval, maxincreasefac, min_epsrel, min_
↪epsabs, max_epsrel, max_epsabs
);
amplitudes.verbose = true;
```

If desired, the contour deformation can be adjusted via additional arguments to `name::handler_t`.

See also:

Section 4.1 and Section 5.9.1 for more detailed information about *name::make_amplitudes()* and `name::handler_t`.

To numerically integrate the sum of sectors, the `name::handler_t::evaluate()` function is called:

```
// compute the amplitudes
const std::vector<box1L::nested_series_t<secdecutil::UncorrelatedDeviation
↪<box1L::integrand_return_t>>> result = amplitudes.evaluate();
```

The remaining lines print the result:

```
// print the result
for (unsigned int amp_idx = 0; amp_idx < box1L::number_of_amplitudes; ++amp_idx)
    std::cout << "amplitude" << amp_idx << " = " << result.at(amp_idx) << std::endl;
```

The C++ program can be built with the command:

```
$ make integrate_box1L
```

A kinematic point must be specified when calling the `integrate_box1L` executable, the input format is:

```
$ ./integrate_box1L 4.0 -0.75 1.25 1.0
```

where the arguments are the `real_parameters` values for (`s`, `t`, `s1`, `msq`). For integrals depending on `complex_parameters`, their value is specified by a space separated pair of numbers representing the real and imaginary part.

If your integral is higher than seven dimensional, changing the integral transform to `integrators::transforms::Baker::type` may improve the accuracy of the result. For further options of the QMC integrator we refer to Section 4.6.2.

2.3 Evaluating a Weighted Sum of Integrals

New in version 1.5.

Let us examine example `easy_sum`, which demonstrates how two weighted sums of dimensionally regulated integrals can be evaluated. The example computes the following two weighted sums:

$$2s\ I_1 + 3s\ I_2,$$
$$\frac{s}{2\epsilon}\ I_1 + \frac{s\epsilon}{3}\ I_2,$$

where

$$I_1 = \int_0^1 \mathrm{d}x \int_0^1 \mathrm{d}y\ (x+y)^{-2+\epsilon},$$
$$I_2 = \int_0^1 \mathrm{d}x \int_0^1 \mathrm{d}y\ (2x+3y)^{-1+\epsilon}.$$

First, we import the necessary python packages and open the `if __name__ == "__main__"` guard, as required by multiprocessing.

```
#!/usr/bin/env python3
from pySecDec import Coefficient
from pySecDec import MakePackage
from pySecDec import sum_package

if __name__ == "__main__":
```

The common arguments for the integrals are collected in the `common_args` dictionary.

```
common_args = {}
common_args['real_parameters'] = ['s']
common_args['regulators'] = ['eps']
common_args['requested_orders'] = [0]
```

Next, the coefficients of the integrals for each weighted sum are specified. Each *Coefficient* is specified as a list of numerator factors, list of denominator factors and a list of real or complex parameters on which the coefficient depends. Coefficients can depend also depend the regulators, the *sum_package* function will automatically determine the correct orders to which the coefficients and integrals should be expanded in order to obtain the `requested_orders`.

```
coefficients = [
    [ # sum1
        Coefficient(['2*s'],['1'],['s']),   # easy1
        Coefficient(['3*s'],['1'],['s'])    # easy2
    ],
    [ # sum2
        Coefficient(['s'],['2*eps'],['s']), # easy1
        Coefficient(['s*eps'],['3'],['s'])  # easy2
    ]
]
```

The integrals are specified using the *MakePackage* wrapper function (which has the same arguments as *make_package*), for loop integrals the *LoopPackage* wrapper may be used (it has the same arguments as *loop_package*).

```
integrals = [
    MakePackage('easy1',
        integration_variables = ['x','y'],
        polynomials_to_decompose = ['(x+y)^(-2+eps)'],
        **common_args),
    MakePackage('easy2',
        integration_variables = ['x','y'],
        polynomials_to_decompose = ['(2*x+3*y)^(-1+eps)'],
        **common_args)
]
```

Finally, the list of integrals and coefficients are passed to *sum_package*. This will generate a C++ library which efficiently evaluates both weighted sums of integrals, sharing the results of the integrals between the different sums.

```
# generate code sum of (int * coeff)
sum_package('easy_sum', integrals,
    coefficients = coefficients, **common_args)
```

The generated C++ library can be *compiled* and called via the *python* and/or *C++* interface as described above.

2.4 Using Expansion By Regions (Generic Integral)

New in version 1.5.

The example `make_regions_ebr` provides a simple introduction to the expansion by regions functionality within pySecDec. For a more detailed discussion of expansion by regions see our paper [PSD21].

The necessary packages are loaded and the `if __name__ == "__main__"` guard is opened.

```
#!/usr/bin/env python3
from pySecDec import sum_package, make_regions

if __name__ == "__main__":
```

Expansion by regions is applied to a generic integral using the *make_regions* function.

```
regions_generators = make_regions(
    name = 'make_regions_ebr',
    integration_variables = ['x'],
    regulators = ['delta'],
    requested_orders = [0],
    smallness_parameter = 't',
    polynomials_to_decompose = ['(x)**(delta)','(t + x + x**2)**(-1)'],
    expansion_by_regions_order = 0,
    real_parameters = ['t'],
    complex_parameters = [],
    decomposition_method = 'geometric_infinity_no_primary',
    polytope_from_sum_of=[1]
)
```

The output of *make_regions* can be passed to *sum_package* in order to generate a C++ library suitable for evaluating the expanded integral.

```
sum_package(
    'make_regions_ebr',
    regions_generators,
    regulators = ['delta'],
    requested_orders = [0],
    real_parameters = ['t']
)
```

The generated C++ library can be *compiled* and called via the *python* and/or *C++* interface as described above.

2.5 Using Expansion By Regions (Loop Integral)

New in version 1.5.

The example `generate_box1L_ebr` demonstrates how expansion by regions can be applied to loop integrals within pySecDec by applying it to the 1-loop box integral as described in Section 4.2 of [Mis18]. For a more detailed discussion of expansion by regions see our paper [PSD21].

First, the necessary packages are loaded and the `if __name__ == "__main__"` guard is opened.

```
#!/usr/bin/env python3

from pySecDec import sum_package, loop_regions
import pySecDec as psd

# This example is the one-loop box example in Go Mishima's paper arXiv:1812.04373

if __name__ == "__main__":
```

The loop integral can be constructed via the convenience functions in *loop_integral*, here we use *LoopintegralFromGraph*. Note that `powerlist=["1+n1","1+n1/2","1+n1/3","1+n1/5"]`, here `n1` is an extra regulator required to regulate the singularities which appear when expanding this loop integral. We use the "trick" of introducing only a single regulator divided by different prime numbers for each power, rather than unique regulators for each propagator (though this is also supported by pySecDec). Poles in the extra regulator `n1` may appear in individual regions but are expected to cancel when all regions are summed.

```
# here we define the Feynman diagram
li = psd.loop_integral.LoopIntegralFromGraph(
internal_lines = [['mt',[3,1]],['mt',[1,2]],['mt',[2,4]],['mt',[4,3]]],
external_lines = [['p1',1],['p2',2],['p3',3],['p4',4]],
powerlist=["1+n1","1+n1/2","1+n1/3","1+n1/5"],
regulators=["eps","n1"],
Feynman_parameters=["x%i" % i for i in range(1,5)], # renames the parameters to get the␣
↪same polynomials as in 1812.04373

replacement_rules = [
                        # note that in those relations all momenta are incoming
                        # general relations:
                        ('p1*p1', 'm1sq'),
                        ('p2*p2', 'm2sq'),
                        ('p3*p3', 'm3sq'),
                        ('p4*p4', 'm4sq'),
```

(continues on next page)

(continued from previous page)

```
                        ('p1*p2', 's/2-(m1sq+m2sq)/2'),
                        ('p1*p3', 't/2-(m1sq+m3sq)/2'),
                        ('p1*p4', 'u/2-(m1sq+m4sq)/2'),
                        ('p2*p3', 'u/2-(m2sq+m3sq)/2'),
                        ('p2*p4', 't/2-(m2sq+m4sq)/2'),
                        ('p3*p4', 's/2-(m3sq+m4sq)/2'),
                        ('u', '(m1sq+m2sq+m3sq+m4sq)-s-t'),
                        # relations for our specific case:
                        ('mt**2', 'mtsq'),
                        ('m1sq',0),
                        ('m2sq',0),
                        ('m3sq','mHsq'),
                        ('m4sq','mHsq'),
                        ('mHsq', 0),
                    ])
```

Expansion by regions is applied to a loop integral using the *loop_regions* function. We expand around a small mass *mtsq*.

```
# find the regions
generators_args = loop_regions(
    name = "box1L_ebr",
    loop_integral=li,
    smallness_parameter = "mtsq",
    expansion_by_regions_order=0)
```

The output of *loop_regions* can be passed to *sum_package* in order to generate a C++ library suitable for evaluating the expanded integral.

```
# write the code to sum up the regions
sum_package("box1L_ebr",
            generators_args,
            li.regulators,
            requested_orders = [0,0],
            real_parameters = ['s','t','u','mtsq'],
            complex_parameters = [])
```

The generated C++ library can be *compiled* and called via the *python* and/or *C++* interface as described above.

2.6 List of Examples

Here we list the available examples. For more details regarding each example see [PSD17], [PSD18] and [PSD21].

easy:	a simple parametric integral, described in Section 2.1
box1L:	a simple 1-loop, 4-point, 4-propagator integral, described in Section 2.2
triangle2L:	a 2-loop, 3-point, 6-propagator diagram, also known as *P126*
box2L_numerator:	massless planar on-shell 2-loop, 4-point, 7-propagator box with a numerator, either defined as an inverse propagator `box2L_invprop.py` or in terms of contracted Lorentz vectors `box2L_contracted_tensor.py`
pentabox_fin:	a 2-loop, 5-point, 8-propagator diagram, evaluated in $6 - 2\epsilon$ dimensions where it is finite
triangle3L:	a 2-loop, 3-point, 7-propagator integral, demonstrates that the symmetry finder can significantly reduce the number of sectors
formfactor4L:	a single-scale 4-loop 3-point integral in $6 - 2\epsilon$ dimensions
bubble6L:	a single-scale 6-loop 2-point integral, evaluated at a Euclidean phase-space point
elliptic2L_euclidean:	an integral known to contain elliptic functions, evaluated at a Euclidean phase-space point
elliptic2L_physical:	an integral known to contain elliptic functions, evaluated at a physical phase-space point
banana_3mass:	a 3-loop 2-point integral with three different internal masses known to contain hyperelliptic functions, evaluated at a physical phase-space point
hyperelliptic:	a 2-loop 4-point nonplanar integral known to contain hyperelliptic functions, evaluated at a physical phase-space point
triangle2L_split:	a 2-loop, 3-point, 6-propagator integral without a Euclidean region due to special kinematics
Nbox2L_split:	three 2-loop, 4-point, 5-propagator integrals that need `split=True` due to special kinematics
hypergeo5F4:	a general dimensionally regulated parameter integral
hz2L_nonplanar:	2-loop, 4-point, 7-propagator integral with internal and external masses
box1L_ebr:	uses expansion by regions to expand a 1-loop box with a small internal mass, this integral is also considered in Section 4.2 of [Mis18]
bubble1L_ebr:	uses expansion by regions to expand a 1-loop, 2-point integral in various limits, demonstrates the use of an additional regulator as described in [PSD21]
bubble1L_dotted_ebr:	uses expansion by regions to expand a 1-loop, 2-point integral, demonstrates the t and z methods described in [PSD21]
bubble2L_largem_ebr:	uses expansion by regions to expand a 1-loop, 2-point integral with a large mass
bubble2L_smallm_ebr:	uses expansion by regions to expand a 1-loop, 2-point integral with a small mass
formfactor1L_ebr:	uses expansion by regions to compute various 1-loop, 3-point form factor integrals from the literature, demonstrates the use of `add_monomial_regulator_power` to introduce an additional regulator as described in [PSD21]
triangle2L_ebr:	uses expansion by regions to compute a 2-loop, 3-point integral with a large mass
make_regions_ebr:	uses expansion by regions to compute a simple generic integral with a small parameter
easy_sum:	calculates the sum of two integrals with different coefficients, demonstrates the use of `sum_package`
yyyy1L:	calculates a 1-loop 4-photon helicity amplitude, demonstrates the use of `sum_package`
two_regulators:	an integral involving poles in two different regulators.
userdefined_cpp:	a collection of examples demonstrating how to combine polynomials to be decomposed with other user-defined functions
regions:	prints a list of the regions obtained by applying expansion by regions to formfactor1L_massless

CHAPTER

THREE

OVERVIEW

pySecDec 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. These are exactly the modules necessary to set up the algebraic computation of a Feynman loop integral requisite for the numerical evaluation. For detailed instruction of a specific function or class, please be referred to the *reference guide*.

3.1 The Algebra Module

The *algebra* module implements a very basic computer algebra system. *pySecDec* uses both *sympy* and *numpy*. 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.

3.1.1 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 and 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 instantiate the `Polynomial` by its algebraic representation:

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

(continues on next page)

(continued from previous page)

```
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.

Within the *Polynomial* class, basic operations are implemented:

```
>>> 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
```

3.1.2 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* to 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 through 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 among 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]
```

3.2 Feynman Parametrization of Loop Integrals

The primary purpose of *pySecDec* is the numerical computation of loop integrals as they arise in fixed order calculations in quantum field theories.

The conventions of *pySecDec* are fixed as follows: a Feynman graph $G^{\mu_1\cdots\mu_R}_{l_1\ldots l_R}$ in D dimensions at L loops with R loop momenta in the numerator and N propagators, where the propagators can have arbitrary, not necessarily integer powers ν_j, is considered to have the following representation in momentum space,

$$G = \int \prod_{l=1}^{L} \mathrm{d}^D\kappa_l \, \frac{k_{l_1}^{\mu_1}\cdots k_{l_R}^{\mu_R}}{\prod_{j=1}^{N} P_j^{\nu_j}(\{k\},\{p\},m_j^2)},$$

$$\mathrm{d}^D\kappa_l = \frac{\mu^{4-D}}{i\pi^{\frac{D}{2}}}\,\mathrm{d}^Dk_l \;,\; P_j(\{k\},\{p\},m_j^2) = (q_j^2 - m_j^2 + i\delta) \;,$$

where the q_j are linear combinations of external momenta p_i and loop momenta k_l.

In the first step of our approach, the loop integral is converted from the momentum representation to the Feynman parameter representation, see for example [Hei08] (Chapter 3).

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

3.2.1 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.

3.2.2 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 +␣
```

(continues on next page)

(continued from previous page)

We can also generate the 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 + (-2*eps*s - 2*eps*t - 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*t + 2*t)*x3**2*x5*x6 + (2*eps*t +␣
↪2*t)*x3**2*x4*x6 + (-2*eps*s - 2*s)*x3**2*x4*x5 + (-2*eps*s - 2*eps*t - 2*s -␣
↪2*t)*x2*x6*U + (-2*eps*s - 2*eps*t - 2*s - 2*t)*x2*x5*U + (-2*eps*s - 2*eps*t - 2*s -␣
↪2*t)*x2*x4*U + (-2*eps*s - 2*eps*t - 2*s - 2*t)*x2*x3*U + (2*eps*t + 2*t)*x2*x3*x6**2␣
↪+ (2*eps*t + 2*t)*x2*x3*x5*x6 + (-2*eps*s + 2*eps*t - 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*t +␣
↪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
```

3.3 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 a variable for sector `p1` evne though it does not depend on it. This declaration is necessary 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]
```

3.4 Subtraction

In the subtraction, we want to perform those integrations that lead to ϵ divergencies. The master formula for one integration variables is

$$\int_0^1 x^{(a-b\epsilon)}\mathcal{I}(x,\epsilon)dx = \sum_{p=0}^{|a|-1} \frac{1}{a+p+1-b\epsilon}\frac{\mathcal{I}^{(p)}(0,\epsilon)}{p!} + \int_0^1 x^{(a-b\epsilon)}R(x,\epsilon)dx$$

where $\mathcal{I}^{(p)}$ is denotes the p-th derivative of $\mathcal{I}$ with respect to x. The equation above effectively defines the remainder term R. All terms on the right hand side of the equation above are constructed to be free of divergencies. For more details and the generalization to multiple variables, we refer the reader to [Hei08]. In the following, we show how to use the implementation in *pySecDec*.

To initialize the subtraction, we first define a factorized expression of the form $x^{(-1-b_x\epsilon)}y^{(-2-b_y\epsilon)}\mathcal{I}(x,y,\epsilon)$:

```
>>> from pySecDec.algebra import Expression
>>> symbols = ['x','y','eps']
>>> x_monomial = Expression('x**(-1 - b_x*eps)', symbols)
>>> y_monomial = Expression('y**(-2 - b_y*eps)', symbols)
>>> cal_I = Expression('cal_I(x, y, eps)', symbols)
```

We must pack the monomials into a *pySecDec.algebra.Product*:

```
>>> from pySecDec.algebra import Product
>>> monomials = Product(x_monomial, y_monomial)
```

Although this seems to be to complete input according to the equation above, we are still missing a structure to store poles in. The function *pySecDec.subtraction.integrate_pole_part()* is designed to return an iterable of the same type as the input. That is particularly important since the output of the subtraction of one variable is the input for the subtraction of the next variable. We will see this iteration later. Initially, we do not have poles yet, therefore we define a *one* of the required type:

```
>>> from pySecDec.algebra import Pow
>>> import numpy as np
```

(continues on next page)

(continued from previous page)

```
>>> polynomial_one = Polynomial(np.zeros([1,len(symbols)], dtype=int), np.array([1]),␣
↪symbols, copy=False)
>>> pole_part_initializer = Pow(polynomial_one, -polynomial_one)
```

`pole_part_initializer` is of type *pySecDec.algebra.Pow* and has `-polynomial_one` in the exponent. We initialize the *base* with `polynomial_one`; i.e. a one packed into a polynomial. The function *pySecDec.subtraction.integrate_pole_part()* populates the *base* with factors of $b\epsilon$ when poles arise.

We are now ready to build the `subtraction_initializer` - the *pySecDec.algebra.Product* to be passed into *pySecDec.subtraction.integrate_pole_part()*.

```
>>> from pySecDec.subtraction import integrate_pole_part
>>> subtraction_initializer = Product(monomials, pole_part_initializer, cal_I)
>>> x_subtracted = integrate_pole_part(subtraction_initializer, 0)
```

The second argument of *pySecDec.subtraction.integrate_pole_part()* specifies to which variable we want to apply the master formula, here we choose x. First, remember that the x monomial is a dimensionally regulated $x^{-}1$. Therefore, the sum collapses to only one term and we have two terms in total. Each term corresponds to one entry in the list `x_subtracted`:

```
>>> len(x_subtracted)
2
```

`x_subtracted` has the same structure as our input. The first factor of each term stores the remaining monomials:

```
>>> x_subtracted[0].factors[0]
(( + (1))**( + (-b_x)*eps + (-1))) * (( + (1)*y)**( + (-b_y)*eps + (-2)))
>>> x_subtracted[1].factors[0]
(( + (1)*x)**( + (-b_x)*eps + (-1))) * (( + (1)*y)**( + (-b_y)*eps + (-2)))
```

The second factor stores the ϵ poles. There is an epsilon pole in the first term, but still none in the second:

```
>>> x_subtracted[0].factors[1]
( + (-b_x)*eps) ** ( + (-1))
>>> x_subtracted[1].factors[1]
( + (1)) ** ( + (-1))
```

The last factor catches everything that is not covered by the first two fields:

```
>>> x_subtracted[0].factors[2]
(cal_I( + (0), + (1)*y, + (1)*eps))
>>> x_subtracted[1].factors[2]
(cal_I( + (1)*x, + (1)*y, + (1)*eps)) + (( + (-1)) * (cal_I( + (0), + (1)*y, + (1)*eps)))
```

We have now performed the subtraction for x. Because in and output have a similar structure, we can easily perform the subtraction for y as well:

```
>>> x_and_y_subtracted = []
>>> for s in x_subtracted:
...     x_and_y_subtracted.extend( integrate_pole_part(s,1) )
```

Alternatively, we can directly instruct *pySecDec.subtraction.integrate_pole_part()* to perform both subtractions:

```
>>> alternative_x_and_y_subtracted = integrate_pole_part(subtraction_initializer,0,1)
```

In both cases, the result is a list of the terms appearing on the right hand side of the master equation.

3.5 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()*).

3.5.1 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`).

3.5.2 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)
```

(continues on next page)

(continued from previous page)

```
  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)))*eps**-1*alpha**-1 + (( + (-1/2)) * (( + (1))**(-
↪1)))*alpha**-2 + (( + (1)) * (( + (2))**(-1)))*eps*alpha**-3
>>> 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

FOUR

SECDECUTIL

SecDecUtil is a standalone autotools-c++ package, that collects common helper classes and functions needed by the c++ code generated using `loop_package` or `make_package`. Everything defined by the *SecDecUtil* is put into the c++ namepace *secdecutil*.

4.1 Amplitude

A collection of utilities for evaluating amplitudes (sums of integrals multiplied by coefficients).

4.1.1 WeightedIntegral

A class template containing an integral, `I`, and the coefficient of the integral, `C`. A `WeightedIntegral` is interpreted as the product `C*I` and can be used to represent individual terms in an amplitude.

template<typename **integral_t**, typename **coefficient_t**>
struct **WeightedIntegral**

std::shared_ptr<*integral_t*> **integral**;
A shared pointer to the integral.

coefficient_t **coefficient**;
The coefficient which will be multiplied on to the integral.

std::string **display_name** = "WINTEGRAL";
A string used to indicate the name of the current weighted integral.

WeightedIntegral(const std::shared_ptr<*integral_t*> &integral, const *coefficient_t* &coefficient = *coefficient_t*(1))

The arithmetic operators (+, -, *, /) are overloaded for `WeightedIntegral` types.

4.1.2 WeightedIntegralHandler

A class template for integrating a sum of `WeightedIntegral` types.

template<typename integrand_return_t, typename real_t, typename coefficient_t, template<typename...> class container_t> class WeightedIntegralHandler

bool **verbose**
: Controls the verbosity of the output of the amplitude.

real_t **min_decrease_factor**
: If the next refinement iteration is expected to make the total time taken for the code to run longer than `wall_clock_limit` then the number of points to be requested in the next iteration will be reduced by at least `min_decrease_factor`.

real_t **decrease_to_percentage**
: If `remaining_time * decrease_to_percentage > time_for_next_iteration` then the number of points requested in the next refinement iteration will be reduced. Here: `remaining_time = wall_clock_limit - elapsed_time` and `time_for_next_iteration` is the estimated time required for the next refinement iteration. Note: if this condition is met this means that the expected precision will not match the desired precision.

real_t **wall_clock_limit**
: If the current elapsed time has passed `wall_clock` limit and a refinement iteration finishes then a new refinement iteration will not be started. Instead, the code will return the current result and exit.

size_t **number_of_threads**
: The number of threads used to compute integrals concurrently. Note: The integrals themselves may also be computed with multiple threads irrespective of this option.

size_t **reset_cuda_after**
: The cuda driver does not automatically remove unnecessary functions from the device memory such that the device may run out of memory after some time. This option controls after how many integrals `cudaDeviceReset()` is called to clear the memory. With the default `0`, `cudaDeviceReset()` is never called. This option is ignored if compiled without cuda.

const container_t<std::vector<term_t>> &**expression**
: The sum of terms to be integrated.

real_t **epsrel**
: The desired relative accuracy for the numerical evaluation of the weighted sum of the sectors.

real_t **epsabs**
: The desired absolute accuracy for the numerical evaluation of the weighted sum of the sectors.

unsigned long long int **maxeval**
: The maximal number of integrand evaluations for each sector.

unsigned long long int **mineval**
: The minimal number of integrand evaluations for each sector.

real_t **maxincreasefac**
: The maximum factor by which the number of integrand evaluations will be increased in a single refinement iteration.

real_t **min_epsrel**
: The minimum relative accuracy required for each individual sector.

real_t **min_epsabs**
: The minimum absolute accuracy required for each individual sector.

real_t **max_epsrel**
: The maximum relative accuracy assumed possible for each individual sector. Any sector known to this precision will not be refined further. Note: if this condition is met this means that the expected precision will not match the desired precision.

real_t **max_epsabs**
: The maximum absolute accuracy assumed possible for each individual sector. Any sector known to this precision will not be refined further. Note: if this condition is met this means that the expected precision will not match the desired precision.

ErrorMode **errormode**
: With `enum ErrorMode : int { abs=0, all, largest, real, imag};`

  Defines how epsrel and epsabs are defined for complex values. With the choice `largest`, the relative uncertainty is defined as `max( |Re(error)|, |Im(error)|)/max( |Re(result)|, |Im(result)|)`. Choosing `all` will apply epsrel and epsabs to both the real and imaginary part separately.

4.2 Series

A class template for containing (optionally truncated) Laurent series. Multivariate series can be represented as series of series.

This class overloads the arithmetic operators (+, -, *, /) and the comparator operators (==, !=). A string representation can be obtained using the << operator. The `at(i)` and `[i]` operators return the coefficient of the i^{th} power of the expansion parameter. Otherwise elements can be accessed identically to `std::vector`.

template<typename **T**>
class **Series**

std::string **expansion_parameter**
: A string representing the expansion parameter of the series (default **x**)

int **get_order_min**() const
: Returns the lowest order in the series.

int **get_order_max**() const
: Returns the highest order in the series.

bool **get_truncated_above**() const
: Checks whether the series is truncated from above.

bool **has_term**(int order) const
: Checks whether the series has a term at order `order`.

Series(int order_min, int order_max, std::vector<*T*> content, bool truncated_above = true, const std::string expansion_parameter = "x")

Example:

```
#include <iostream>
#include <secdecutil/series.hpp>

int main()
{
    secdecutil::Series<int> exact(-2,1,{1,2,3,4},false,"eps");
```

(continues on next page)

(continued from previous page)

```
    secdecutil::Series<int> truncated(-2,1,{1,2,3,4},true,"eps");
    secdecutil::Series<secdecutil::Series<int>> multivariate(1,2,
                                                             {
                                                                 {-2,-1,{1,2},false,
↪"alpha"},
                                                                 {-2,-1,{3,4},false,
↪"alpha"},
                                                             },false,"eps"
                                                             );

    std::cout << "exact:         " << exact << std::endl;
    std::cout << "truncated:     " << truncated << std::endl;
    std::cout << "multivariate: " << multivariate << std::endl << std::endl;

    std::cout << "exact + 1:          " << exact + 1 << std::endl;
    std::cout << "exact * exact:      " << exact * exact << std::endl;
    std::cout << "exact * truncated: " << exact * truncated << std::endl;
    std::cout << "exact.at(-2):       " << exact.at(-2) << std::endl;
}
```

Compile/Run:

```
$ c++ -I${SECDEC_CONTRIB}/include -std=c++14 example.cpp -o example -lm && ./example
```

Output:

```
exact:         + (1)*eps^-2 + (2)*eps^-1 + (3) + (4)*eps
truncated:     + (1)*eps^-2 + (2)*eps^-1 + (3) + (4)*eps + O(eps^2)
multivariate:  + ( + (1)*alpha^-2 + (2)*alpha^-1)*eps + ( + (3)*alpha^-2 + (4)*alpha^-
↪1)*eps^2

exact + 1:          + (1)*eps^-2 + (2)*eps^-1 + (4) + (4)*eps
exact * exact:      + (1)*eps^-4 + (4)*eps^-3 + (10)*eps^-2 + (20)*eps^-1 + (25) +␣
↪(24)*eps + (16)*eps^2
exact * truncated:  + (1)*eps^-4 + (4)*eps^-3 + (10)*eps^-2 + (20)*eps^-1 + O(eps^0)
exact.at(-2):       1
```

4.3 Deep Apply

A general concept to apply a `std::function` to a nested data structure. If the applied `std::function` is not void then *deep_apply()* returns a nested data structure of the return values. Currently *secdecutil* implements this for `std::vector` and *Series*.

This concept allows, for example, the elements of a nested series to be edited without knowing the depth of the nested structure.

> template<typename **Tout**, typename **Tin**, template<typename...> class **Tnest**>
> *Tnest*<*Tout*> **deep_apply**(*Tnest*<*Tin*> &nest, std::function<*Tout*(*Tin*)> &func)

Example (complex conjugate a *Series*):

```cpp
#include <iostream>
#include <complex>
#include <secdecutil/series.hpp>
#include <secdecutil/deep_apply.hpp>

int main()
{
    std::function<std::complex<double>(std::complex<double>)> conjugate =
    [] (std::complex<double> element)
    {
        return std::conj(element);
    };

    secdecutil::Series<std::complex<double>> u(-1,0,{{1,2},{3,4}},false,"eps");
    secdecutil::Series<secdecutil::Series<std::complex<double>>> m(1,1,{{1,1,{{1,2}},
↪false,"alpha"},},false,"eps");

    std::cout << "u: " << u << std::endl;
    std::cout << "m: " << m << std::endl << std::endl;

    std::cout << "conjugated u:   " << secdecutil::deep_apply(u, conjugate) << std::endl;
    std::cout << "conjugated m: " << secdecutil::deep_apply(m, conjugate) << std::endl;
}
```

Compile/Run:

```
$ c++ -I${SECDEC_CONTRIB}/include -std=c++14 example.cpp -o example -lm && ./example
```

Output:

```
u:  + ((1,2))*eps^-1 + ((3,4))
m:  + ( + ((1,2))*alpha)*eps

conjugated u:    + ((1,-2))*eps^-1 + ((3,-4))
conjugated m:  + ( + ((1,-2))*alpha)*eps
```

4.4 Uncertainties

A class template which implements uncertainty propagation for uncorrelated random variables by overloads of the `+`, `-`, `*` and partially `/`. Division by *UncorrelatedDeviation* is not implemented as it is not always defined. It has special overloads for `std::complex<T>`.

Note: Division by *UncorrelatedDeviation* is not implemented as this operation is not always well defined. Specifically, it is ill defined in the case that the errors are Gaussian distributed as the expectation value,

$$\mathrm{E}\left[\frac{1}{X}\right] = \int_{-\infty}^{\infty} \frac{1}{X} p(X)\,\mathrm{d}X,$$

where

$$p(X) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right),$$

is undefined in the Riemann or Lebesgue sense. The rule $\delta(a/b) = |a/b|\sqrt{(\delta a/a)^2 + (\delta b/b)^2}$ can not be derived from the first principles of probability theory.

The rules implemented for real valued error propagation are:

$$\delta(a+b) = \sqrt{(\delta a)^2 + (\delta b)^2},$$

$$\delta(a-b) = \sqrt{(\delta a)^2 + (\delta b)^2},$$

$$\delta(ab) = \sqrt{(\delta a)^2 b^2 + (\delta b)^2 a^2 + (\delta a)^2 (\delta b)^2}.$$

For complex numbers the above rules are implemented for the real and imaginary parts individually.

template<typename **T**>
class **UncorrelatedDeviation**

T **value**
The expectation value.

T **uncertainty**
The standard deviation.

Example:

```
#include <iostream>
#include <complex>
#include <secdecutil/uncertainties.hpp>

int main()
{
    secdecutil::UncorrelatedDeviation<double> r(1.,0.5);
    secdecutil::UncorrelatedDeviation<std::complex<double>> c({2.,3.},{0.6,0.7});

    std::cout << "r: " << r << std::endl;
    std::cout << "c: " << c << std::endl << std::endl;

    std::cout << "r.value:       " << r.value << std::endl;
    std::cout << "r.uncertainty: " << r.uncertainty << std::endl;
    std::cout << "r + c:         " << r + c << std::endl;
    std::cout << "r * c:         " << r * c << std::endl;
    std::cout << "r / 3.0:       " << r / 3. << std::endl;
    // std::cout << "1. / r:      " << 1. / r << std::endl; // ERROR
    // std::cout << "c / r:       " << c / r << std::endl;  // ERROR
}
```

Compile/Run:

```
$ c++ -I${SECDEC_CONTRIB}/include -std=c++14 example.cpp -o example -lm && ./example
```

Output:

```
r: 1 +/- 0.5
c: (2,3) +/- (0.6,0.7)

r.value:       1
```

(continues on next page)

(continued from previous page)

```
r.uncertainty: 0.5
r + c:         (3,3) +/- (0.781025,0.7)
r * c:         (2,3) +/- (1.20416,1.69189)
r / 3.0:       0.333333 +/- 0.166667
```

4.5 Integrand Container

A class template for containing integrands. It stores the number of integration variables and the integrand as a `std::function`.

This class overloads the arithmetic operators (+, -, *, /) and the call operator (()).

template<typename **T**, typename ...**Args**>
class **IntegrandContainer**

int **number_of_integration_variables**
The number of integration variables that the integrand depends on.

std::function<*T*(*Args*...)> **integrand**
The integrand function. The call operator forwards to this function.

Example (add two *IntegrandContainer* and evaluate one point):

```
#include <iostream>
#include <secdecutil/integrand_container.hpp>

int main()
{
    using input_t = const double * const;
    using return_t = double;

    const std::function<return_t(input_t, secdecutil::ResultInfo*)> f1 = [] (input_t x,␣
↪secdecutil::ResultInfo* result_info) { return 2*x[0]; };
    secdecutil::IntegrandContainer<return_t,input_t> c1(1,f1);

    const std::function<return_t(input_t, secdecutil::ResultInfo*)> f2 = [] (input_t x,␣
↪secdecutil::ResultInfo* result_info) { return x[0]*x[1]; };
    secdecutil::IntegrandContainer<return_t,input_t> c2(2,f2);

    secdecutil::IntegrandContainer<return_t,input_t> c3 = c1 + c2;

    const double point[]{1.0,2.0};
    const double parameters[]{};
    secdecutil::ResultInfo* result_info;

    std::cout << "c1.number_of_integration_variables: " << c1.number_of_integration_
↪variables << std::endl;
    std::cout << "c2.number_of_integration_variables: " << c2.number_of_integration_
↪variables << std::endl << std::endl;
    std::cout << "c3.number_of_integration_variables: " << c3.number_of_integration_
↪variables << std::endl;
```

(continues on next page)

(continued from previous page)

```
    std::cout << "c3.integrand(point, parameters, result_info): " << c3.integrand_with_
↪parameters(point, parameters, result_info) << std::endl;
}
```

Compile/Run:

```
$ c++ -I${SECDEC_CONTRIB}/include -std=c++14 example.cpp -o example -lm && ./example
```

Output:

```
c1.number_of_integration_variables: 1
c2.number_of_integration_variables: 2

c3.number_of_integration_variables: 2
c3.integrand(point, parameters, result_info): 4
```

4.6 Integrator

A base class template from which integrator implementations inherit. It defines the minimal API available for all integrators.

template<typename **return_t**, typename **input_t**, typename **container_t** =
secdecutil::*IntegrandContainer*<*return_t*, *input_t* const*const>>
class **Integrator**

bool **together**
(Only available if `return_t` is a `std::complex` type) If `true` after each call of the function both the real and imaginary parts are passed to the underlying integrator. If `false` after each call of the function only the real or imaginary part is passed to the underlying integrator. For some adaptive integrators considering the real and imaginary part of a complex function separately can improve the sampling. Default: `false`.

UncorrelatedDeviation<*return_t*> **integrate**(const *IntegrandContainer*<*return_t*, *input_t* const*const>&)
Integrates the *IntegrandContainer* and returns the value and uncertainty as an *UncorrelatedDeviation*.

An integrator that chooses another integrator based on the dimension of the integrand.

template<typename **return_t**, typename **input_t**>
class **MultiIntegrator**

Integrator<*return_t*, *input_t*> &**low_dimensional_integrator**
Reference to the integrator to be used if the integrand has a lower dimension than *critical_dim*.

Integrator<*return_t*, *input_t*> &**high_dimensional_integrator**
Reference to the integrator to be used if the integrand has dimension *critical_dim* or higher.

int **critical_dim**
The dimension below which the *low_dimensional_integrator* is used.

4.6.1 CQuad

For one dimensional integrals, we wrap the cquad integrator form the GNU scientifc library (gsl).

CQuad takes the following options:

- `epsrel` - The desired relative accuracy for the numerical evaluation. Default: `0.01`.
- `epsabs` - The desired absolute accuracy for the numerical evaluation. Default: `1e-7`.
- `n` - The size of the workspace. This value can only be set in the constructor. Changing this attribute of an instance is not possible. Default: `100`.
- `verbose` - Whether or not to print status information. Default: `false`.
- `zero_border` - The minimal value an integration variable can take. Default: `0.0`. (*new in version 1.3*)

4.6.2 Qmc

The quasi-monte carlo integrator as described in [PSD18]. Using a quasi-monte integrator to compute sector decomposed integrals was pioneered in [LWY+15].

template<typename **`return_t`**, ::integrators::U **`maxdim`**, template<typename, typename, ::integrators::U> class **`transform_t`**, typename **`container_t`** = secdecutil::*IntegrandContainer*<*return_t*, typename remove_complex<*return_t*>::type const*const>, template<typename, typename, ::integrators::U> class **`fitfunction_t`** = void_template>
class **`Qmc`** : *Integrator*<*return_t*, *return_t*, *container_t*>, public ::integrators::Qmc<*return_t*, *return_t*, *maxdim*, *transform_t*, *fitfunction_t*>

Derived from *`secdecutil::Integrator`* and `::integrators::Qmc` - the underlying standalone implementation of the Qmc.

The most important fields and template argments of *`Qmc`* are:

- `minn` - The minimal number of points in the Qmc lattice. Will be augmented to the next larger available `n`.
- `minm` - The minimal number of random shifts.
- `maxeval` - The maximal number of integrand evaluations.
- `epsrel` - The desired relative accuracy for the numerical evaluation.
- `epsabs` - The desired absolute accuracy for the numerical evaluation.
- `maxdim` - The highest dimension the *`Qmc`* instance can be used for.
- `transform_t` - The periodizing transform to apply prior to integration.
- `fitfunction_t` - The fit function transform to apply for adaptive integration.
- `verbosity` - Controls the amount of status messages during integration. Can be `0`, `1`, `2`, or `3`.
- `devices` - A `std::set` of devices to run on. `-1` denotes the CPU, positive integers refer to GPUs.

Refer to the documentation of the standalone Qmc for the default values and additional information.

An integral transform has to be chosen by setting the template argument `transform_t`. Available transforms are e.g. `Korobov<r0,r1>` and `Sidi<r0>`, please refer to the underlying Qmc implementation for a complete list. The fit function for adaptive integration can be set by the `fitfunction_t`, e.g. `PolySingular`. If not set, the default of the underlying Qmc implementation is used.

Examples how to use the Qmc *on the CPU* and on *both, CPU and GPU* are shown below.

4.6.3 Cuba

Currently we wrap the following Cuba integrators:

- `Vegas`
- `Suave`
- `Divonne`
- `Cuhre`

The Cuba integrators all implement:

- `epsrel` - The desired relative accuracy for the numerical evaluation. Default: `0.01`.
- `epsabs` - The desired absolute accuracy for the numerical evaluation. Default: `1e-7`.
- `flags` - Sets the Cuba verbosity flags. The `flags=2` means that the Cuba input parameters and the result after each iteration are written to the log file of the numerical integration. Default: `0`.
- `seed` - The seed used to generate random numbers for the numerical integration with Cuba. Default: `0`.
- `mineval` - The number of evaluations which should at least be done before the numerical integrator returns a result. Default: `0`.
- `maxeval` - The maximal number of evaluations to be performed by the numerical integrator. Default: `1000000`.
- `zero_border` - The minimal value an integration variable can take. Default: `0.0`. (*new in version 1.3*)

The available integrator specific parameters and their default values are:

Vegas	Suave	Divonne	Cuhre
nstart (`10000`)	nnew (`1000`)	key1 (`2000`)	key (`0`)
nincrease (`5000`)	nmin (`10`)	key2 (1)	
nbatch (`500`)	flatness (`25.0`)	key3 (1)	
		maxpass (4)	
		border (`0.0`)	
		maxchisq (`1.0`)	
		mindeviation (`0.15`)	

For the description of these more specific parameters we refer to the Cuba manual.

4.6.4 Examples

Integrate Real Function with Cuba Vegas

Example:

```
#include <iostream>
#include <secdecutil/integrand_container.hpp>
#include <secdecutil/uncertainties.hpp>
#include <secdecutil/integrators/cuba.hpp>

int main()
{
    using input_t = const double * const;
```

(continues on next page)

(continued from previous page)

```
    using return_t = double;

    secdecutil::cuba::Vegas<return_t> integrator;
    integrator.epsrel = 1e-4;
    integrator.maxeval = 1e7;

    secdecutil::IntegrandContainer<return_t,input_t> c(2, [] (input_t x,␣
↪secdecutil::ResultInfo* result_info) { return x[0]*x[1]; });
    secdecutil::UncorrelatedDeviation<return_t> result = integrator.integrate(c);

    std::cout << "result: " << result << std::endl;
}
```

Compile/Run:

```
$ c++ -I${SECDEC_CONTRIB}/include -L${SECDEC_CONTRIB}/lib -std=c++14 example.cpp -o␣
↪example -lcuba -lm && ./example
```

Output:

```
result: 0.250004 +/- 2.43875e-05
```

Integrate Complex Function with Cuba Vegas

Example:

```
#include <iostream>
#include <complex>
#include <secdecutil/integrand_container.hpp>
#include <secdecutil/uncertainties.hpp>
#include <secdecutil/integrators/cuba.hpp>

int main()
{
    using input_t = const double * const;
    using return_t = std::complex<double>;

    secdecutil::cuba::Vegas<return_t> integrator;
    const std::function<return_t(input_t, secdecutil::ResultInfo*)> f = [] (input_t x,␣
↪secdecutil::ResultInfo* result_info) { return return_t{x[0],x[1]}; };
    secdecutil::IntegrandContainer<return_t,input_t> c(2,f);

    integrator.together = false; // integrate real and imaginary part separately␣
↪(default)
    secdecutil::UncorrelatedDeviation<return_t> result_separate = integrator.
↪integrate(c);

    integrator.together = true; // integrate real and imaginary part simultaneously
    secdecutil::UncorrelatedDeviation<return_t> result_together = integrator.
↪integrate(c);
```

(continues on next page)

(continued from previous page)

```
    std::cout << "result_separate: " << result_separate << std::endl;
    std::cout << "result_together: " << result_together << std::endl;

}
```

Compile/Run:

```
$ c++ -I${SECDEC_CONTRIB}/include -L${SECDEC_CONTRIB}/lib -std=c++14 example.cpp -o␣
↪example -lcuba -lm && ./example
```

Output:

```
result_separate: (0.499937,0.499937) +/- (0.00288675,0.00288648)
result_together: (0.499937,0.499937) +/- (0.00288675,0.00288648)
```

Integrate Real Function with Cuba Vegas or CQuad

Example:

```
#include <iostream>
#include <secdecutil/integrand_container.hpp>
#include <secdecutil/uncertainties.hpp>
#include <secdecutil/integrators/integrator.hpp>
#include <secdecutil/integrators/cuba.hpp>
#include <secdecutil/integrators/cquad.hpp>

int main()
{
    using input_base_t = double;
    using input_t = const input_base_t * const;
    using return_t = double;

    secdecutil::cuba::Vegas<return_t> vegas;
    vegas.epsrel = 1e-5;
    vegas.maxeval = 1e7;

    secdecutil::gsl::CQuad<return_t> cquad;
    cquad.epsrel = 1e-10;
    cquad.epsabs = 1e-13;

    secdecutil::MultiIntegrator<return_t,input_base_t> integrator(cquad,vegas,2);

    secdecutil::IntegrandContainer<return_t,input_t> one_dimensional(1, [] (input_t x,␣
↪secdecutil::ResultInfo* result_info) { return x[0]; });
    secdecutil::IntegrandContainer<return_t,input_t> two_dimensional(2, [] (input_t x,␣
↪secdecutil::ResultInfo* result_info) { return x[0]*x[1]; });

    secdecutil::UncorrelatedDeviation<return_t> result_1d = integrator.integrate(one_
↪dimensional); // uses cquad
    secdecutil::UncorrelatedDeviation<return_t> result_2d = integrator.integrate(two_
↪dimensional); // uses vegas
```

(continues on next page)

(continued from previous page)

```
    std::cout << "result_1d: " << result_1d << std::endl;
    std::cout << "result_2d: " << result_2d << std::endl;
}
```

Compile/Run:

```
$ c++ -I${SECDEC_CONTRIB}/include -L${SECDEC_CONTRIB}/lib -std=c++14 example.cpp -o␣
↪example -lcuba -lgsl -lgslcblas -lm && ./example
```

Output:

```
result_1d: 0.5 +/- 9.58209e-17
result_2d: 0.25 +/- 5.28257e-06
```

Set the integral transform of the Qmc

Example:

```
#include <iostream>
#include <secdecutil/integrand_container.hpp>
#include <secdecutil/uncertainties.hpp>
#include <secdecutil/integrators/qmc.hpp>

using input_base_t = double;
using input_t = input_base_t const * const;
using return_t = double;
using container_t = secdecutil::IntegrandContainer<return_t,input_t>;
using result_t = secdecutil::UncorrelatedDeviation<return_t>;

const int seed = 12345, maxdim = 4;

int main()
{
    /*
     * minimal instantiation
     */
    secdecutil::integrators::Qmc
    <
        return_t, // the return type of the integrand
        maxdim, // the highest dimension this integrator will be used for
        ::integrators::transforms::Baker::type // the integral transform
    > integrator_baker;
    integrator_baker.randomgenerator.seed(seed);

    /*
     * disable adaptation
     */
    secdecutil::integrators::Qmc
    <
        return_t, // the return type of the integrand
```

(continues on next page)

(continued from previous page)

```
        maxdim, // the highest dimension this integrator will be used for
        ::integrators::transforms::Korobov<4,1>::type, // the integral transform
        container_t, // the functor type to be passed to this integrator
        ::integrators::fitfunctions::None::type // the fit funtion
    > integrator_korobov4x1;
    integrator_korobov4x1.randomgenerator.seed(seed);

    /*
     * enable adaptation
     */
    secdecutil::integrators::Qmc
    <
        return_t, // the return type of the integrand
        maxdim, // the highest dimension this integrator will be used for
        ::integrators::transforms::Sidi<3>::type, // the integral transform
        container_t, // the functor type to be passed to this integrator
        ::integrators::fitfunctions::PolySingular::type // the fit funtion
    > integrator_sidi3_adaptive;
    integrator_sidi3_adaptive.randomgenerator.seed(seed);

    // define the integrand as a functor
    container_t integrand(
                            4, // dimension
                            [] (input_t x, secdecutil::ResultInfo* result_info) {␣
↪return x[0]*x[1]*x[2]*x[3]; } // integrand function
                          );

    // compute the integral with different settings
    result_t result_baker = integrator_baker.integrate(integrand);
    result_t result_korobov4x1 = integrator_korobov4x1.integrate(integrand);
    result_t result_sidi3_adaptive = integrator_sidi3_adaptive.integrate(integrand);

    // print the results
    std::cout << "baker: " << result_baker << std::endl;
    std::cout << "Korobov (weights 4, 1): " << result_korobov4x1 << std::endl;
    std::cout << "Sidi (weight 3, adaptive): " << result_sidi3_adaptive << std::endl;
}
```

Compile/Run:

```
c++ -I${SECDEC_CONTRIB}/include -pthread -L${SECDEC_CONTRIB}/lib -std=c++14 example.cpp -
↪o example -lm && ./example
```

Output:

```
baker: 0.0625 +/- 7.93855e-08
Korobov (weights 4, 1): 0.0625108 +/- 2.97931e-05
Sidi (weight 3, adaptive): 0.0625 +/- 4.33953e-09
```

Run the Qmc on GPUs

Example:

```
#include <iostream>
#include <secdecutil/integrand_container.hpp>
#include <secdecutil/uncertainties.hpp>
#include <secdecutil/integrators/qmc.hpp>

using input_base_t = double;
using input_t = input_base_t const * const;
using return_t = double;
using container_t = secdecutil::IntegrandContainer<return_t,input_t>;
using result_t = secdecutil::UncorrelatedDeviation<return_t>;

/*
 * `container_t` cannot be used on the GPU --> define a different container type
 */
struct cuda_integrand_t
{
    const static unsigned number_of_integration_variables = 4;

    // integrand function
    #ifdef __CUDACC__
        __host__ __device__
    #endif
    return_t operator()(input_t x)
    {
        return x[0]*x[1]*x[2]*x[3];
    };

    void process_errors() const{ /* error handling */}
} cuda_integrand;

const int seed = 12345, maxdim = 4;

int main()
{
    /*
     * Qmc capable of sampling on the GPU
     */
    secdecutil::integrators::Qmc
    <
        return_t, // the return type of the integrand
        maxdim, // the highest dimension this integrator will be used for
        ::integrators::transforms::Sidi<3>::type, // the integral transform
        cuda_integrand_t, // the functor type to be passed to this integrator
        ::integrators::fitfunctions::PolySingular::type // the fit funtion (optional)
    > integrator_sidi3_adaptive_gpu;
    integrator_sidi3_adaptive_gpu.randomgenerator.seed(seed);

    // compute the integral with different settings
    result_t result_sidi3_adaptive_gpu = integrator_sidi3_adaptive_gpu.integrate(cuda_
↪integrand);
```

(continues on next page)

(continued from previous page)

```
    // print the results
    std::cout << "Sidi (weight 3, adaptive): " << result_sidi3_adaptive_gpu << std::endl;
}
```

Compile/Run:

```
nvcc -x cu -I${SECDEC_CONTRIB}/include -L${SECDEC_CONTRIB}/lib -std=c++14 example.cpp -o␣
↪example -lgsl -lgslcblas -lm && ./example # with GPU
c++ -I${SECDEC_CONTRIB}/include -pthread -L${SECDEC_CONTRIB}/lib -std=c++14 example.cpp -
↪o example -lgsl -lgslcblas -lm && ./example # without GPU
```

Output:

```
Sidi (weight 3, adaptive): 0.0625 +/- 4.33953e-09
```

CHAPTER

FIVE

REFERENCE GUIDE

This section describes all public functions and classes in *pySecDec*.

5.1 Algebra

Implementation of a simple computer algebra system.

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.

static from_expression(**args*, ***kwargs*)

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']".

refactorize(**parameters*)
Returns a product of the greatest factor that could be pulled out and the factorised polynomial.

Parameters **parameter** – arbitrarily many integers;

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

pySecDec.algebra.**Expression**(*expression*, *polysymbols*, *follow_functions=False*)
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.
- **follow_functions** – bool, optional (default = `False`); If true, return the converted expression and a list of *Function* that occur in the *expression*.

class pySecDec.algebra.**Function**(*symbol*, **arguments*, ***kwargs*)
Symbolic function that can take care of parameter transformations. It keeps track of all taken derivatives: When *derive()* is called, save the multiindex of the taken derivative.

The derivative multiindices are the keys in the dictionary `self.derivative_tracks`. 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, Function
>>> x = Polynomial.from_expression('x', ['x','y'])
>>> y = Polynomial.from_expression('y', ['x','y'])
>>> poly = x**2*y + y**2
>>> func = Function('f', x, y)
>>> ddfuncd0d1 = func.derive(0).derive(1)
>>> func
Function(f( + (1)*x, + (1)*y), derivative_tracks = {(1, 0): [0, (0, 0)], (1, 1): [1,
↪ (1, 0)]})
>>> func.derivative_tracks
{(1, 0): [0, (0, 0)], (1, 1): [1, (1, 0)]}
>>> func.compute_derivatives(poly)
{(1, 0):  + (2)*x*y, (1, 1):  + (2)*x}
```

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*.

compute_derivatives(*expression=None*)
Compute all derivatives of `expression` that are mentioned in `self.derivative_tracks`. The purpose of this function is to avoid computing the same derivatives multiple times.

Parameters **expression** – _Expression, optional; The expression to compute the derivatives of. If not provided, the derivatives are shown as in terms of the *function*'s derivatives dfd<index>.

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 dfd<index>.

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()
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(*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(*indices=None*)
Return True if the polynomial can be written as:

$$const + ...$$

Otherwise, return False.

Parameters **indices** – list of integers or None; The indices of the *polysymbols* to consider. If `None` (default) all indices are taken into account.

refactorize(**parameters*)
Returns a product of the greatest factor that could be pulled out and the factorised polynomial.

Parameters **parameter** – arbitrarily many integers;

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(*deep=True*)
Combine terms that have the same exponents of the variables.

Parameters **deep** – bool; If `True` (default) call the *simplify* method of the coefficients if they are of type `_Expression`.

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(*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> or 1**<something> = 1 if possible. Convert to *ExponentiatedPolynomial* or *Polynomial* 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(*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*.

to_sum()
Convert the *ProductRule* to *Sum*

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(*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.

pySecDec.algebra.**refactorize**(*polyprod*, **parameters*)
In a *algebra.Product* of the form <*monomial*> * <*polynomial*>, check if a parameter in <*polynomial*> can be shifted to the <*monomial*>. If possible, modify *polyprod* accordingly.

Parameters

- **polyprod** – *algebra.Product* of the form <monomial> * <polynomial>`; The product to refactorize.
- **parameter** – integer, optional; Check only the parameter with this index. If not provided, all parameters are checked.

5.2 Loop Integral

This module defines routines to Feynman parametrize a loop integral and build a c++ package that numerically integrates over the sector decomposed integrand.

5.2.1 Feynman Parametrization

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.Gamma_factor`
- `self.exponentiated_U`
- `self.exponentiated_F`
- `self.numerator`
- `self.measure`,

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

When inverse propagators or nonnumerical propagator powers are present (see *powerlist*), some *Feynman_parameters* drop out of the integral. The variables to integrate over can be accessed as `self.integration_variables`.

While `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`.

Changed in version 1.2.2: The overall sign $(-1)^{N_\nu}$ is included in `self.Gamma_factor`.

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'*, *regulators=None*, *regulator=None*, *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)
```

(continues on next page)

(continued from previous page)

```
>>> 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.
- **regulators** – list of strings or sympy symbol, optional; The symbols to be used for the regulators (typically ϵ or ϵ_D)

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

- **regulator** – a string or a sympy symbol, optional; Deprecated; same as setting *regulators* to a list of a single element.
- **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 *regulators*. In case of negative powers, the *numerator* is constructed by taking derivatives with respect to the corresponding Feynman parameters as explained in Section 3.2.4 of Ref. [BHJ+15]. If negative powers are combined with a tensor numerator, the derivatives act on the Feynman-parametrized tensor numerator as well, which leads 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'*, *regulators=None*, *regulator=None*, *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.

  See also:

  parameter *numerator*

- **Lorentz_indices** – iterable of strings or sympy expressions, optional; Symbols to be used as Lorentz indices in the numerator.

  See also:

  parameter *numerator*

- **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 must be a sum of products of exclusively:

  – numbers

  – scalar products (e.g. `p1(mu)*k1(mu)*p1(nu)*k2(nu)`)

  – symbols (e.g. `s`, `eps`)

  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)`

Note: In order to use the resulting *LoopIntegral* as an argument to the function *pySecDec.loop_integral.loop_package()*, the resulting Feynman parametrized `self.numerator` must be expressible as a *pySecDec.algebra.Polynomial* such that all coefficients are purely numeric. In addition, all scalar products of the external momenta must be expressed in terms of Mandelstam variables using the *replacement_rules*.

Warning: **All** Lorentz indices (including the contracted ones and also including the numbers that have been used) must be explicitly defined using the parameter *Lorentz_indices*.

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(mu)`. If indices are left open, however, the LoopIntegral cannot be used with the package generator *pySecDec.loop_integral.loop_package()*.

- **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.
- **regulators** – list of strings or sympy symbol, optional; The symbols to be used for the regulators (typically ϵ or ϵ_D)

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

- **regulator** – a string or a sympy symbol, optional; Deprecated; same as setting *regulators* to a list of a single element.
- **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 *regulators*. In case of negative powers, the *numerator* is constructed by taking derivatives with respect to the corresponding Feynman parameters as explained in Section 3.2.4 of Ref. [BHJ+15]. If negative powers are combined with a tensor numerator, the derivatives act on the Feynman-parametrized tensor numerator as well, which leads to a consistent result.

5.2.2 Loop Package

This module contains the function that generates a c++ package.

pySecDec.loop_integral.**loop_package**(*name, loop_integral, requested_orders=None, requested_order=None, real_parameters=[], complex_parameters=[], contour_deformation=True, additional_prefactor=1, form_optimization_level=2, form_work_space='50M', form_memory_use=None, form_threads=2, decomposition_method='iterative', normaliz_executable='normaliz', enforce_complex=False, split=False, ibp_power_goal=-1, use_iterative_sort=True, use_light_Pak=True, use_dreadnaut=False, use_Pak=True, processes=None, pylink_qmc_transforms=['korobov3x3'], package_generator=<function make_package>*)

Decompose, subtract and expand a Feynman parametrized loop integral. Return it as c++ package.

See also:

This function is a wrapper around *pySecDec.make_package()* (default).

See also:

The generated library is described in *Generated C++ Libraries*.

Parameters

- **name** – string; The name of the c++ namespace and the output directory.
- **loop_integral** – *pySecDec.loop_integral.LoopIntegral*; The loop integral to be computed.
- **requested_orders** – iterable of integers; Compute the expansion in the regulators to these orders.
- **requested_order** – integer; Deprecated; same as *requested_orders* set to a list of one item.
- **real_parameters** – iterable of strings or sympy symbols, optional; Parameters to be interpreted as real numbers, e.g. Mandelstam invariants and masses.
- **complex_parameters** – iterable of strings or sympy symbols, optional; Parameters to be interpreted as complex numbers. To use the complex mass scheme, define the masses as complex parameters.
- **contour_deformation** – bool, optional; Whether or not to produce code for contour deformation. Default: `True`.
- **additional_prefactor** – string or sympy expression, optional; An additional factor to be multiplied to the loop integral. It may depend on the regulators, the *real_parameters*, and the *complex_parameters*.
- **form_optimization_level** – integer out of the interval [0,4], optional; The optimization level to be used in FORM. Default: `2`.
- **form_work_space** – string, optional; The FORM WorkSpace. Default: `'50M'`.

  Setting this to smaller values will reduce FORM memory usage (without affecting performance), but each problem has some minimum value below which FORM will refuse to work: it will fail with error message indicating that larger WorkSpace is needed, at which point WorkSpace will be adjusted and FORM will be re-run.

- **form_memory_use** – string, optional; The target FORM memory usage. When specified, *form.set* parameters will be adjusted so that FORM uses at most approximately this much resident memory.

  The minimum is approximately to 600M + 350M per worker thread if `form_work_space` is left at `'50M'`. if form_work_space is increased to `'500M'`, then the minimum is 2.5G + 2.5G per worker thread. Default: `None`, meaning use the default FORM values.

- **form_threads** – integer, optional; Number of threads (T)FORM will use. Default: `2`.

- **decomposition_method** – string, optional; The strategy for decomposing the polynomials. The following strategies are available:

  – 'iterative' (default)

  – 'geometric'

  – 'geometric_ku'

  Note: For 'geometric' and 'geometric_ku', the third-party program "normaliz" is needed. See *The Geomethod and Normaliz*.

- **normaliz_executable** – string, optional; The command to run *normaliz*. *normaliz* is only required if *decomposition_method* is set to 'geometric' or 'geometric_ku'. Default: 'normaliz'

- **enforce_complex** – bool, optional; Whether or not the generated integrand functions should have a complex return type even though they might be purely real. The return type of the integrands is automatically complex if *contour_deformation* is `True` or if there are *complex_parameters*. In other cases, the calculation can typically be kept purely real. Most commonly, this flag is needed if `log(<negative real>)` occurs in one of the integrand functions. However, *pySecDec* will suggest setting this flag to `True` in that case. Default: `False`

- **split** – bool, optional; Whether or not to split the integration domain in order to map singularities from 1 to 0. Set this option to `True` if you have singularties when one or more integration variables are one. Default: `False`

- **ibp_power_goal** – number or iterable of number, optional; The *power_goal* that is forwarded to *integrate_by_parts()*.

  This option controls how the subtraction terms are generated. Setting it to `-numpy.inf` disables *integrate_by_parts()*, while `0` disables *integrate_pole_part()*.

  See also:

  To generate the subtraction terms, this function first calls *integrate_by_parts()* for each integration variable with the give *ibp_power_goal*. Then *integrate_pole_part()* is called.

  Default: `-1`

- **use_iterative_sort** – bool; Whether or not to use *squash_symmetry_redundant_sectors_sort()* with *iterative_sort()* to find sector symmetries. Default: `True`

- **use_light_Pak** – bool; Whether or not to use *squash_symmetry_redundant_sectors_sort()* with *light_Pak_sort()* to find sector symmetries. Default: `True`

- **use_dreadnaut** – bool or string, optional; Whether or not to use *squash_symmetry_redundant_sectors_dreadnaut()* to find sector symmetries. If

given a string, interpret that string as the command line executable *dreadnaut*. If `True`, try `$SECDEC_CONTRIB/bin/dreadnaut` and, if the environment variable `$SECDEC_CONTRIB` is not set, `dreadnaut`. Default: `False`

- **use_Pak** – bool; Whether or not to use *squash_symmetry_redundant_sectors_sort()* with *Pak_sort()* to find sector symmetries. Default: `True`
- **processes** – integer or None, optional; The maximal number of processes to be used. If `None`, the number of CPUs `multiprocessing.cpu_count()` is used. *New in version 1.3*. Default: `None`
- **pylink_qmc_transforms** – list or None, optional; Required qmc integral transforms, options are:
  - `korobov<i>x<j>` for 1 <= i,j <= 6
  - `korobov<i>` for 1 <= i <= 6 (same as `korobov<i>x<i>`)
  - `sidi<i>` for 1 <= i <= 6

  New in version 1.5. Default: `['korobov3x3']`
- **package_generator** – function; The generator function for the integral, either *pySecDec.make_package()* or *pySecDec.code_writer.make_package()*.

  Default: *pySecDec.make_package()*.

5.2.3 Drawing Feynman Diagrams

Use the following function to draw Feynman diagrams.

pySecDec.loop_integral.draw.**plot_diagram**(*internal_lines*, *external_lines*, *filename*, *powerlist=None*, *neato='neato'*, *extension='pdf'*, *Gstart=0*)

Draw a Feynman diagram using Graphviz (neato).

Thanks to Viktor Papara <papara@mpp.mpg.de> for his major contributions to this function.

Note: This function requires the command line tool *neato*. See also *Drawing Feynman Diagrams with neato*.

Warning: The target is overwritten without prompt if it exists already.

Parameters

- **internal_lines** – list; Adjacency list of internal lines, e.g. `[['m',['a',4]],['m',[4,5]], ['m',['a',5]],[0,[1,2]],[0,[4,1]],[0,[2,5]]]`
- **external_lines** – list; Adjacency list of external lines, e.g. [['p1',1],['p2',2],['p3','a']]
- **filename** – string; The name of the output file. The generated file gets this name plus the file *extension*.
- **powerlist** – list, optional; The powers of the propagators defined by the *internal_lines*.
- **neato** – string, default: "neato"; The shell command to call "neato".
- **extension** – string, default: "pdf"; The file extension. This also defines the output format.
- **Gstart** – nonnegative int; The is value is passed to "neato" with the "-Gstart" option. Try changing this value if the visualization looks bad.

5.2.4 Loop Regions

Applies the expansion by regions method to a loop integral.

pySecDec.loop_integral.**loop_regions**(*name*, *loop_integral*, *smallness_parameter*, *expansion_by_regions_order=0*, *contour_deformation=True*, *additional_prefactor=1*, *form_optimization_level=2*, *form_work_space='500M'*, *add_monomial_regulator_power=None*, *decomposition_method='iterative'*, *normaliz_executable='normaliz'*, *enforce_complex=False*, *split=False*, *ibp_power_goal=- 1*, *use_iterative_sort=True*, *use_light_Pak=True*, *use_dreadnaut=False*, *use_Pak=True*, *processes=None*)

Apply expansion by regions method to the loop integral using the function.

See also:

This function is a wrapper around *pySecDec.make_regions()*.

See also:

The generated library is described in *Generated C++ Libraries*.

Parameters

- **name** – string; The name of the c++ namespace and the output directory.
- **loop_integral** – *pySecDec.loop_integral*; The loop integral to which the expansion by regions method is applied.
- **smallness_parameter** – string or sympy symbol; The symbol of the variable in which the expression is expanded.
- **expansion_by_regions_order** – integer; The order up to which the expression is expanded in the *smallness_parameter*. Default: 0
- **contour_deformation** – bool, optional; Whether or not to produce code for contour deformation. Default: `True`.
- **additional_prefactor** – string or sympy expression, optional; An additional factor to be multiplied to the loop integral. It may depend on the regulators, the *real_parameters*, and the *complex_parameters*.
- **form_optimization_level** – integer out of the interval [0,4], optional; The optimization level to be used in FORM. Default: `2`.
- **form_work_space** – string, optional; The FORM WorkSpace. Default: `'50M'`.

  Setting this to smaller values will reduce FORM memory usage (without affecting performance), but each problem has some minimum value below which FORM will refuse to work: it will fail with error message indicating that larger WorkSpace is needed, at which point WorkSpace will be adjusted and FORM will be re-run.
- **add_monomial_regulator_power** – string or sympy symbol; Name of the regulator, using which monomial factors of the form $x_i**(n/p_i)$ are added, to regulate the integrals arising from the expansion by regions.
- **decomposition_method** – string, optional; The strategy for decomposing the polynomials. The following strategies are available:
  - 'iterative' (default)
  - 'geometric'

– 'geometric_ku'

Note: For 'geometric' and 'geometric_ku', the third-party program "normaliz" is needed. See *The Geomethod and Normaliz*.

- **normaliz_executable** – string, optional; The command to run *normaliz*. *normaliz* is only required if *decomposition_method* is set to 'geometric' or 'geometric_ku'. Default: 'normaliz'
- **enforce_complex** – bool, optional; Whether or not the generated integrand functions should have a complex return type even though they might be purely real. The return type of the integrands is automatically complex if *contour_deformation* is `True` or if there are *complex_parameters*. In other cases, the calculation can typically be kept purely real. Most commonly, this flag is needed if `log(<negative real>)` occurs in one of the integrand functions. However, *pySecDec* will suggest setting this flag to `True` in that case. Default: `False`
- **split** – bool, optional; Whether or not to split the integration domain in order to map singularities from 1 to 0. Set this option to `True` if you have singularties when one or more integration variables are one. Default: `False`
- **ibp_power_goal** – number or iterable of number, optional; The *power_goal* that is forwarded to *integrate_by_parts()*.

  This option controls how the subtraction terms are generated. Setting it to `-numpy.inf` disables *integrate_by_parts()*, while `0` disables *integrate_pole_part()*.

  See also:

  To generate the subtraction terms, this function first calls *integrate_by_parts()* for each integration variable with the give *ibp_power_goal*. Then *integrate_pole_part()* is called.

  Default: `-1`
- **use_iterative_sort** – bool; Whether or not to use *squash_symmetry_redundant_sectors_sort()* with *iterative_sort()* to find sector symmetries. Default: `True`
- **use_light_Pak** – bool; Whether or not to use *squash_symmetry_redundant_sectors_sort()* with *light_Pak_sort()* to find sector symmetries. Default: `True`
- **use_dreadnaut** – bool or string, optional; Whether or not to use *squash_symmetry_redundant_sectors_dreadnaut()* to find sector symmetries. If given a string, interpret that string as the command line executable *dreadnaut*. If `True`, try `$SECDEC_CONTRIB/bin/dreadnaut` and, if the environment variable `$SECDEC_CONTRIB` is not set, `dreadnaut`. Default: `False`
- **use_Pak** – bool; Whether or not to use *squash_symmetry_redundant_sectors_sort()* with *Pak_sort()* to find sector symmetries. Default: `True`
- **processes** – integer or None, optional; The maximal number of processes to be used. If `None`, the number of CPUs `multiprocessing.cpu_count()` is used. *New in version 1.3*. Default: `None`

5.3 Polytope

The polytope class as required by *pySecDec.make_regions* and *pySecDec.decomposition.geometric*.

class pySecDec.polytope.**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]. See *The Geomethod and Normaliz* for installation and a list of tested versions.

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.polytope.**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.polytope.**triangulate**(*cone*, *normaliz='normaliz'*, *workdir='normaliz_tmp'*, *keep_workdir=False*, *switch_representation=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* [BIR]. See *The Geomethod and Normaliz* for installation and a list of tested versions.

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.
- **switch_representation** – bool; Whether or not to switch between facet and vertex/ray representation.

5.4 Decomposition

The core of sector decomposition. This module implements the actual decomposition routines.

5.4.1 Common

This module collects routines that are used by multiple decompition modules.

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.**squash_symmetry_redundant_sectors_sort**(*sectors*, *sort_function*, *indices=None*)

Reduce a list of sectors by squashing duplicates with equal integral.

If two sectors only differ by a permutation of the polysymbols (to be interpreted as integration variables over some inteval), then the two sectors integrate to the same value. Thus we can drop one of them and count the other twice. The multiple counting of a sector is accounted for by increasing the coefficient of the Jacobian by one.

Equivalence up to permutation is established by applying the *sort_function* to each sector, this brings them into a canonical form. Sectors with identical canonical forms differ only by a permutation.

Note: whether all symmetries are found depends on the choice of *sort_function*. The sort function *pySecDec.matrix_sort.Pak_sort()* should find all symmetries whilst the sort functions *pySecDec.matrix_sort.iterative_sort()* and *pySecDec.matrix_sort.light_Pak_sort()* are faster but do not identify all symmetries.

See also: *squash_symmetry_redundant_sectors_dreadnaut()*

Example:

```
>>> from pySecDec.algebra import Polynomial
>>> from pySecDec.decomposition import Sector
>>> from pySecDec.decomposition import squash_symmetry_redundant_sectors_sort
>>> from pySecDec.matrix_sort import Pak_sort
>>>
>>> poly = Polynomial([(0,1),(1,0)], ['a','b'])
>>> swap = Polynomial([(1,0),(0,1)], ['a','b'])
>>> Jacobian_poly = Polynomial([(1,0)], [3]) # three
>>> Jacobian_swap = Polynomial([(0,1)], [5]) # five
>>> sectors = (
...               Sector([poly],Jacobian=Jacobian_poly),
...               Sector([swap],Jacobian=Jacobian_swap)
...           )
>>>
>>> reduced_sectors = squash_symmetry_redundant_sectors_sort(sectors,
...                   Pak_sort)
>>> len(reduced_sectors) # symmetry x0 <--> x1
1
>>> # The Jacobians are added together to account
>>> # for the double counting of the sector.
>>> reduced_sectors[0].Jacobian
 + (8)*x0
```

Parameters

- **sectors** – iterable of *Sector*; the sectors to be reduced.
- **sort_function** – *pySecDec.matrix_sort.iterative_sort()*, *pySecDec.matrix_sort.light_Pak_sort()*, or *pySecDec.matrix_sort.Pak_sort()*; The function to be used for finding a canonical form of the sectors.
- **indices** – iterable of integers, optional; The indices of the variables to consider. If not provided, all indices are taken into account.

pySecDec.decomposition.**squash_symmetry_redundant_sectors_dreadnaut**(*sectors*, *indices=None*, *dreadnaut='dreadnaut'*, *workdir='dreadnaut_tmp'*, *keep_workdir=False*)

Reduce a list of sectors by squashing duplicates with equal integral.

Each *Sector* is converted to a *Polynomial* which is represented as a graph following the example of [MP+14] (v2.6 Figure 7, Isotopy of matrices).

We first multiply each polynomial in the sector by a unique tag then sum the polynomials of the sector, this converts a sector to a polynomial. Next, we convert the *expolist* of the resulting polynomial to a graph where each unique exponent in the *expolist* is considered to be a different symbol. Each unique coefficient in the polynomial`s *coeffs* is assigned a vertex and connected to the row vertex of any term it multiplies. The external program *dreadnaut* is then used to bring the graph into a canonical form and provide a hash. Sectors with

equivalent hashes may be identical, their canonical graphs are compared and if they are identical the sectors are combined.

Note: This function calls the command line executable of *dreadnaut* [MP+14]. It has been tested with *dreadnaut* version nauty26r7.

See also: *squash_symmetry_redundant_sectors_sort()*

Parameters

- **sectors** – iterable of *Sector*; the sectors to be reduced.
- **indices** – iterable of integers, optional; The indices of the variables to consider. If not provided, all indices are taken into account.
- **dreadnaut** – string; The shell command to run *dreadnaut*.
- **workdir** – string; The directory for the communication with *dreadnaut*. 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 *dreadnaut* is done via files.

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

5.4.2 Iterative

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.**find_singular_set**(*sector*, *indices=None*)
Function within the iterative sector decomposition procedure which heuristically chooses an optimal decomposition set. The strategy was introduced in arXiv:hep-ph/0004013 [BH00] and is described in 4.2.2 of arXiv:1410.7939 [Bor14]. Return a list of indices.

Parameters

- **sector** – *Sector*; The sector to be decomposed.
- **indices** – iterable of integers or None; The indices of the parameters to be considered as integration variables. By default (`indices=None`), all parameters are considered as integration variables.

pySecDec.decomposition.iterative.**iteration_step**(*sector*, *indices=None*)
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.
- **indices** – iterable of integers or None; The indices of the parameters to be considered as integration variables. By default (`indices=None`), all parameters are considered as integration variables.

pySecDec.decomposition.iterative.**iterative_decomposition**(*sector*, *indices=None*)
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.
- **indices** – iterable of integers or None; The indices of the parameters to be considered as integration variables. By default (`indices=None`), all parameters are considered as integration variables.

pySecDec.decomposition.iterative.**primary_decomposition**(*sector*, *indices=None*)
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.
- **indices** – iterable of integers or None; The indices of the parameters to be considered as integration variables. By default (`indices=None`), all parameters are considered as integration variables.

pySecDec.decomposition.iterative.**primary_decomposition_polynomial**(*polynomial*, *indices=None*)
Perform the primary decomposition on a single polynomial.

See also:

primary_decomposition()

Parameters

- **polynomial** – *algebra.Polynomial*; The polynomial to eliminate the Dirac delta from.
- **indices** – iterable of integers or None; The indices of the parameters to be considered as integration variables. By default (`indices=None`), all parameters are considered as integration variables.

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*; 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)
```

5.4.3 Geometric

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)

pySecDec.decomposition.geometric.**generate_fan**(**polynomials*)

Calculate the fan of the polynomials in the input. The rays of a cone are given by the exponent vectors after factoring out a monomial together with the standard basis vectors. Each choice of factored out monomials gives a different cone. Only full (N-) dimensional cones in $R^N_{\geq 0}$ need to be considered.

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

pySecDec.decomposition.geometric.**geometric_decomposition**(*sector*, *indices=None*, *normaliz='normaliz'*, *workdir='normaliz_tmp'*)

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

Note: This function calls the command line executable of *normaliz* [BIR]. See *The Geomethod and Normaliz* for installation and a list of tested versions.

Parameters

- **sector** – *Sector*; The sector to be decomposed.
- **indices** – list of integers or None; The indices of the parameters to be considered as integration variables. By default (`indices=None`), all parameters are considered as integration variables.
- **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.**geometric_decomposition_ku**(*sector*, *indices=None*, *normaliz='normaliz'*, *workdir='normaliz_tmp'*)

Run the sector decomposition using the original geometric decomposition strategy by Kaneko and Ueda as described in [KU10].

Note: This function calls the command line executable of *normaliz* [BIR]. See *The Geomethod and Normaliz* for installation and a list of tested versions.

Parameters

- **sector** – *Sector*; The sector to be decomposed.
- **indices** – list of integers or None; The indices of the parameters to be considered as integration variables. By default (`indices=None`), all parameters are considered as integration variables.
- **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*,

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

, 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.

5.4.4 Splitting

Routines to split the integration between 0 and 1. This maps singularities from 1 to 0.

pySecDec.decomposition.splitting.**find_singular_sets_at_one**(*polynomial*)

Find all possible sets of parameters such that the *polynomial*'s constant term vanishes if these parameters are set to one.

Example:

```
>>> from pySecDec.algebra import Polynomial
>>> from pySecDec.decomposition.splitting import find_singular_sets_at_one
>>> polysymbols = ['x0', 'x1']
>>> poly = Polynomial.from_expression('1 - 10*x0 - x1', polysymbols)
>>> find_singular_sets_at_one(poly)
[(1,)]
```

Parameters **polynomial** – *Polynomial*; The polynomial to search in.

pySecDec.decomposition.splitting.**remap_one_to_zero**(*polynomial*, **indices*)

Apply the transformation $x \to 1 - x$ to *polynomial* for the parameters of the given *indices*.

Parameters

- **polynomial** – *Polynomial*; The polynomial to apply the transformation to.
- **indices** – arbitrarily many int; The indices of the `polynomial.polysymbols` to apply the transformation to.

Example:

```
>>> from pySecDec.algebra import Polynomial
>>> from pySecDec.decomposition.splitting import remap_one_to_zero
>>> polysymbols = ['x0']
>>> polynomial = Polynomial.from_expression('x0', polysymbols)
>>> remap_one_to_zero(polynomial, 0)
 + (1) + (-1)*x0
```

pySecDec.decomposition.splitting.**split**(*sector*, *seed*, **indices*)

Split the integration interval $[0, 1]$ for the parameters given by *indices*. The splitting point is fixed using *numpy's* random number generator.

Return an iterator of *Sector* - the arising subsectors.

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

:param seed; integer; The seed for the random number generator that is used to fix the splitting point.

Parameters **indices** – arbitrarily many integers; The indices of the variables to be split.

pySecDec.decomposition.splitting.**split_singular**(*sector*, *seed*, *indices=[]*)

Split the integration interval $[0, 1]$ for the parameters that can lead to singularities at one for the polynomials in `sector.cast`.

Return an iterator of *Sector* - the arising subsectors.

Parameters

- **sector** – *Sector*; The sector to be split.
- **seed** – integer; The seed for the random number generator that is used to fix the splitting point.
- **indices** – iterables of integers; The indices of the variables to be split if required. An empty iterator means that all variables may potentially be split.

5.5 Matrix Sort

Algorithms to sort a matrix when column and row permutations are allowed.

pySecDec.matrix_sort.**Pak_sort**(*matrix*, **indices*)

Inplace modify the *matrix* to some canonical ordering, when permutations of rows and columns are allowed.

The *indices* parameter can contain a list of lists of column indices. Only the columns present in the same list are swapped with each other.

The implementation of this function is described in chapter 2 of [Pak11].

Note: If not all indices are considered the resulting matrix may not be canonical.

See also:

iterative_sort(), *light_Pak_sort()*

Parameters

- **matrix** – 2D array-like; The matrix to be canonicalized.
- **indices** – arbitrarily many iterables of non-negative integers; The groups of columns to permute. Default: `range(1,matrix.shape[1])`

pySecDec.matrix_sort.**iterative_sort**(*matrix*)

Inplace modify the *matrix* to some ordering, when permutations of rows and columns (excluding the first) are allowed.

Note: This function may result in different orderings depending on the initial ordering.

See also:

Pak_sort(), *light_Pak_sort()*

Parameters **matrix** – 2D array-like; The matrix to be canonicalized.

pySecDec.matrix_sort.**light_Pak_sort**(*matrix*)

Inplace modify the *matrix* to some ordering, when permutations of rows and columns (excluding the first) are allowed. The implementation of this function is described in chapter 2 of [Pak11]. This function implements a lightweight version: In step (v), we only consider one, not all table copies with the minimized second column.

Note: This function may result in different orderings depending on the initial ordering.

See also:

iterative_sort(), *Pak_sort()*

Parameters **matrix** – 2D array-like; The matrix to be canonicalized.

5.6 Subtraction

Routines to isolate the divergencies in an ϵ expansion.

pySecDec.subtraction.**integrate_by_parts**(*polyprod*, *power_goals*, *indices*)

Repeatedly apply integration by parts,

$$\int_0^1 dt_j t_j^{(a_j - b_j\epsilon_1 - c\epsilon_2 + \ldots)} \mathcal{I}(t_j, \{t_{i\neq j}\}, \epsilon_1, \epsilon_2, \ldots) = \frac{1}{a_j + 1 - b_j\epsilon_1 - c\epsilon_2 - \ldots} \left(\mathcal{I}(1, \{t_{i\neq j}\}, \epsilon_1, \epsilon_2, \ldots) - \int_0^1 dt_j t_j^{(a_j + 1 - b_j\epsilon_1 - c\epsilon_2 +}$$

, where $\mathcal{I}'$ denotes the derivative of $\mathcal{I}$ with respect to t_j. The iteration stops, when $a_j >=$ *power_goal_j*.

See also:

This function provides an alternative to *integrate_pole_part()*.

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 $\epsilon \to 0$ 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*.
- **power_goals** – number or iterable of numbers, e.g. float, integer, ...; The stopping criterion for the iteration.
- **indices** – iterable of 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. Each returned list element has the same structure as the input *polyprod*.

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

Transform an integral of the form

$$\int_0^1 dt_j t_j^{(a - b\epsilon_1 - c\epsilon_2 + \ldots)} \mathcal{I}(t_j, \{t_{i\neq j}\}, \epsilon_1, \epsilon_2, \ldots)$$

into the form

$$\sum_{p=0}^{|a|-1} \frac{1}{a + p + 1 - b\epsilon_1 - c\epsilon_2 - \ldots} \frac{\mathcal{I}^{(p)}(0, \{t_{i\neq j}\}, \epsilon_1, \epsilon_2, \ldots)}{p!} + \int_0^1 dt_j t_j^{(a - b\epsilon_1 - c\epsilon_2 + \ldots)} R(t_j, \{t_{i\neq j}\}, \epsilon_1, \epsilon_2, \ldots)$$

, where $\mathcal{I}^{(p)}$ 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 $\epsilon_1, \epsilon_2, ...$ 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 $\epsilon \to 0$ 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*.

pySecDec.subtraction.**pole_structure**(*monomial_product*, **indices*)

Return a list of the unregulated exponents of the parameters specified by *indices* in *monomial_product*.

Parameters

- **monomial_product** – *pySecDec.algebra.ExponentiatedPolynomial* with `exponent` being a *Polynomial*; The monomials of the subtraction to extract the pole structure from.
- **indices** – arbitrarily many integers; The index/indices of the parameter(s) to partially investigate.

5.7 Expansion

Routines to series expand singular and nonsingular expressions.

exception pySecDec.expansion.**OrderError**

This exception is raised if an expansion to a lower than the lowest order of an expression is requested.

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

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

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

See also:

To expand more general expressions use *expand_sympy()*.

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.

pySecDec.expansion.**expand_sympy**(*expression*, *variables*, *orders*)
Expand a sympy expression in the *variables* to given *orders*. Return the expansion as nested *pySecDec.algebra.Polynomial*.

See also:

This function is a generalization of *expand_singular()*.

Parameters

- **expression** – string or sympy expression; The expression to be expanded
- **variables** – iterable of strings or sympy symbols; The variables to expand the *expression* in.
- **orders** – iterable of integers; The orders to expand to.

5.8 Code Writer

This module collects routines to create a c++ library.

5.8.1 Make Package

This is the main function of *pySecDec*.

pySecDec.code_writer.**make_package**(*name*, *integration_variables*, *regulators*, *requested_orders*, *polynomials_to_decompose*, *polynomial_names=[]*, *other_polynomials=[]*, *prefactor=1*, *remainder_expression=1*, *functions=[]*, *real_parameters=[]*, *complex_parameters=[]*, *form_optimization_level=2*, *form_work_space='50M'*, *form_memory_use=None*, *form_threads=2*, *form_insertion_depth=5*, *contour_deformation_polynomial=None*, *positive_polynomials=[]*, *decomposition_method='iterative_no_primary'*, *normaliz_executable='normaliz'*, *enforce_complex=False*, *split=False*, *ibp_power_goal=-1*, *use_iterative_sort=True*, *use_light_Pak=True*, *use_dreadnaut=False*, *use_Pak=True*, *processes=None*, *pylink_qmc_transforms=['korobov3x3']*)

Decompose, subtract and expand an expression. Return it as c++ package.

See also:

In order to decompose a loop integral, use the function *pySecDec.loop_integral.loop_package()*.

See also:

The generated library is described in *Generated C++ Libraries*.

Parameters

- **name** – string; The name of the c++ namepace and the output directory.
- **integration_variables** – iterable of strings or sympy symbols; The variables that are to be integrated. The intgration region depends on the chosen *decomposition_method*.
- **regulators** – iterable of strings or sympy symbols; The UV/IR regulators of the integral.
- **requested_orders** – iterable of integers; Compute the expansion in the regulators to these orders.
- **polynomials_to_decompose** – iterable of strings or sympy expressions or *pySecDec.algebra.ExponentiatedPolynomial* or *pySecDec.algebra.Polynomial*; The polynomials to be decomposed.
- **polynomial_names** – iterable of strings; Assign symbols for the *polynomials_to_decompose*. These can be referenced in the *other_polynomials*; see *other_polynomials* for details.
- **other_polynomials** – iterable of strings or sympy expressions or *pySecDec.algebra.ExponentiatedPolynomial* or *pySecDec.algebra.Polynomial*; Additional polynomials where no decomposition is attempted. The symbols defined in *polynomial_names* can be used to reference the *polynomials_to_decompose*. This is particularly useful when computing loop integrals where the "numerator" can depend on the first and second Symanzik polynomials.

  Example (1-loop bubble with numerator):

```
>>> polynomials_to_decompose = ["(x0 + x1)**(2*eps - 4)",
...                             "(-p**2*x0*x1)**(-eps))"]
>>> polynomial_names = ["U", "F"]
>>> other_polynomials = """   (eps - 1)*s*U**2
...                         + (eps - 2)*F
...                         - (eps - 1)*2*s*x0*U
...                         + (eps - 1)*s*x0**2"""]
```

See also:

pySecDec.loop_integral

Note that the *polynomial_names* refer to the *polynomials_to_decompose* **without** their exponents.

- **prefactor** – string or sympy expression, optional; A factor that does not depend on the integration variables.
- **remainder_expression** – string or sympy expression or `pySecDec.algebra._Expression`, optional; An additional factor.

  Dummy function must be provided with all arguments, e.g. `remainder_expression='exp(eps)*f(x0,x1)'`. In addition, all dummy function must be listed in *functions*.
- **functions** – iterable of strings or sympy symbols, optional; Function symbols occuring in *remainder_expression*, e.g.``['f']``.

Note: Only user-defined functions that are provided as c++-callable code should be mentioned here. Listing basic mathematical functions (e.g. `log`, `pow`, `exp`, `sqrt`, ...) is not required and considered an error to avoid name conflicts.

Note: The power function *pow* and the logarithm *log* use the nonstandard continuation with an infinitesimal negative imaginary part on the negative real axis (e.g. `log(-1) = -i*pi`).

- **real_parameters** – iterable of strings or sympy symbols, optional; Symbols to be interpreted as real variables.
- **complex_parameters** – iterable of strings or sympy symbols, optional; Symbols to be interpreted as complex variables.
- **form_optimization_level** – integer out of the interval [0,4], optional; The optimization level to be used in FORM. Default: `2`.
- **form_work_space** – string, optional; The FORM WorkSpace. Default: `'50M'`.

  Setting this to smaller values will reduce FORM memory usage (without affecting performance), but each problem has some minimum value below which FORM will refuse to work: it will fail with error message indicating that larger WorkSpace is needed, at which point WorkSpace will be adjusted and FORM will be re-run.
- **form_memory_use** – string, optional; The target FORM memory usage. When specified, *form.set* parameters will be adjusted so that FORM uses at most approximately this much resident memory.

  The minimum is approximately to 600M + 350M per worker thread if `form_work_space` is left at `'50M'`. if form_work_space is increased to `'500M'`, then the minimum is 2.5G + 2.5G per worker thread. Default: `None`, meaning use the default FORM values.
- **form_threads** – integer, optional; Number of threads (T)FORM will use. Default: `2`.
- **form_insertion_depth** – nonnegative integer, optional; How deep FORM should try to resolve nested function calls. Default: `5`.
- **contour_deformation_polynomial** – string or sympy symbol, optional; The name of the polynomial in *polynomial_names* that is to be continued to the complex plane according

to a $-i\delta$ prescription. For loop integrals, this is the second Symanzik polynomial `F`. If not provided, no code for contour deformation is created.

- **positive_polynomials** – iterable of strings or sympy symbols, optional; The names of the polynomials in *polynomial_names* that should always have a positive real part. For loop integrals, this applies to the first Symanzik polynomial `U`. If not provided, no polynomial is checked for positiveness. If *contour_deformation_polynomial* is `None`, this parameter is ignored.
- **decomposition_method** – string, optional; The strategy to decompose the polynomials. The following strategies are available:
  - 'iterative_no_primary' (default): integration region $[0,1]^N$.
  - 'geometric_no_primary': integration region $[0,1]^N$.
  - 'geometric_infinity_no_primary': integration region $[0,\infty]^N$.
  - 'iterative': primary decomposition followed by integration over $[0,1]^{N-1}$.
  - 'geometric': x_N is set to one followed by integration over $[0,\infty]^{N-1}$.
  - 'geometric_ku': primary decomposition followed by integration over $[0,1]^{N-1}$.

  'iterative', 'geometric', and 'geometric_ku' are only valid for loop integrals. An end user should use 'iterative_no_primary', 'geometric_no_primary', or 'geometric_infinity_no_primary' here. In order to compute loop integrals, please use the function *pySecDec.loop_integral.loop_package()*.
- **normaliz_executable** – string, optional; The command to run *normaliz*. *normaliz* is only required if *decomposition_method* starts with 'geometric'. Default: 'normaliz'
- **enforce_complex** – bool, optional; Whether or not the generated integrand functions should have a complex return type even though they might be purely real. The return type of the integrands is automatically complex if *contour_deformation* is `True` or if there are *complex_parameters*. In other cases, the calculation can typically be kept purely real. Most commonly, this flag is needed if `log(<negative real>)` occurs in one of the integrand functions. However, *pySecDec* will suggest setting this flag to `True` in that case. Default: `False`
- **split** – bool or integer, optional; Whether or not to split the integration domain in order to map singularities from 1 to 0. Set this option to `True` if you have singularties when one or more integration variables are one. If an integer is passed, that integer is used as seed to generate the splitting point. Default: `False`
- **ibp_power_goal** – number or iterable of number, optional; The *power_goal* that is forwarded to *integrate_by_parts()*.

  This option controls how the subtraction terms are generated. Setting it to `-numpy.inf` disables *integrate_by_parts()*, while `0` disables *integrate_pole_part()*.

  See also:

  To generate the subtraction terms, this function first calls *integrate_by_parts()* for each integration variable with the give *ibp_power_goal*. Then *integrate_pole_part()* is called.

  Default: `-1`
- **use_iterative_sort** – bool; Whether or not to use *squash_symmetry_redundant_sectors_sort()* with *iterative_sort()* to find sector symmetries. Default: `True`

- **use_light_Pak** – bool; Whether or not to use *squash_symmetry_redundant_sectors_sort()* with *light_Pak_sort()* to find sector symmetries. Default: `True`
- **use_dreadnaut** – bool or string, optional; Whether or not to use *squash_symmetry_redundant_sectors_dreadnaut()* to find sector symmetries. If given a string, interpret that string as the command line executable *dreadnaut*. If `True`, try `$SECDEC_CONTRIB/bin/dreadnaut` and, if the environment variable `$SECDEC_CONTRIB` is not set, `dreadnaut`. Default: `False`
- **use_Pak** – bool; Whether or not to use *squash_symmetry_redundant_sectors_sort()* with *Pak_sort()* to find sector symmetries. Default: `True`
- **processes** – integer or None, optional; Parallelize the package generation using at most this many processes. If `None`, use the total number of logical CPUs on the system (that is, `os.cpu_count()`), or the number of CPUs allocated to the current process (`len(os.sched_getaffinity(0))`), on platforms where this information is available (i.e. Linux+glibc). *New in version 1.3*. Default: `None`
- **pylink_qmc_transforms** – list or None, optional; Required qmc integral transforms, options are:
  - `korobov<i>x<j>` for 1 <= i,j <= 6
  - `korobov<i>` for 1 <= i <= 6 (same as `korobov<i>x<i>`)
  - `sidi<i>` for 1 <= i <= 6

  New in version 1.5. Default: `['korobov3x3']`

5.8.2 Sum Package

Computing weighted sums of integrals, e.g. amplitudes.

class pySecDec.code_writer.sum_package.**Coefficient**(*numerators*, *denominators*, *parameters*)

Store a coefficient expressed as a product of terms in the numerator and a product of terms in the denominator.

Parameters

- **numerators** – iterable of str; The terms in the numerator.
- **denominators** – iterable of str; The terms in the denominator.
- **parameters** – iterable of strings or sympy symbols; The symbols other parameters.

process(*regulators*, *form=None*, *workdir='form_tmp'*, *keep_workdir=False*)

Calculate and return the lowest orders of the coefficients in the regulators and a string defining the expressions "numerator", "denominator", and "regulator_factor".

Parameters

- **regulators** – iterable of strings or sympy symbols; The symbols denoting the regulators.
- **form** – string or None; If given a string, interpret that string as the command line executable *form*. If `None`, try `$FORM` (if the environment variable `$FORM` is set), `$SECDEC_CONTRIB/bin/form` (if `$SECDEC_CONTRIB` is set), and `form`.
- **workdir** – string; The directory for the communication with *form*. 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 *form* is done via files.

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

pySecDec.code_writer.sum_package.**sum_package**(*name*, *package_generators*, *regulators*, *requested_orders*, *real_parameters=[]*, *complex_parameters=[]*, *coefficients=None*, *form_executable=None*, *pylink_qmc_transforms=['korobov3x3']*, *processes=1*)

Decompose, subtract and expand a list of expressions of the form

$$\sum_j c_{ij} \int f_j$$

Generate a c++ package with an optmized algorithm to evaluate the integrals numerically. It writes the names of the integrals in the file *"integral_names.txt"*. For the format of the file see `Parser`.

Parameters

- **name** – string; The name of the c++ namepace and the output directory.
- **package_generators** – iterable of *pySecDec.code_writer.MakePackage* and/or *pySecDec.loop_integral.LoopPackage namedtuples*; The generator functions for the integrals $\int f_i$

  Note: The *pySecDec.code_writer.MakePackage* and *pySecDec.loop_integral.LoopPackage* objects have the same argument list as their respective parent functions *pySecDec.code_writer.make_package()* and *pySecDec.loop_integral.loop_package()*.

- **regulators** – iterable of strings or sympy symbols; The UV/IR regulators of the integral.
- **requested_orders** – iterable of integers; Compute the expansion in the *regulators* to these orders.
- **real_parameters** – iterable of strings or sympy symbols, optional; Symbols to be interpreted as real variables.
- **complex_parameters** – iterable of strings or sympy symbols, optional; Symbols to be interpreted as complex variables.
- **coefficients** – iterable of iterable of *Coefficient*, optional; The coefficients c_{ij} of the integrals. $c_{ij} = 1$ with $i \in \{0\}$ is assumed if not provided.
- **form_executable** – string or None, optional; The path to the form exectuable. The argument is passed to *Coefficient.process()*. If `None`, then either `$FORM`, `$SECDEC_CONTRIB/bin/form`, or just `form` is used, depending on which environment variable is set. Default: `None`.
- **pylink_qmc_transforms** – list or None, optional; Required qmc integral transforms, options are:
  - `korobov<i>x<j>` for 1 <= i,j <= 6
  - `korobov<i>` for 1 <= i <= 6 (same as `korobov<i>x<i>`)
  - `sidi<i>` for 1 <= i <= 6

  Default: `['korobov3x3']`
- **processes** – integer or None, optional; Parallelize the generation of terms in a sum using this many processes.

  When set to a value larger than 1, this will override the `processes` argument of the terms in a sum, meaning that each term will not use parallelization, but rather different terms will be generated in parallel.

Default: 1

5.8.3 Template Parser

Functions to generate c++ sources from template files.

pySecDec.code_writer.template_parser.**generate_pylink_qmc_macro_dict**(*macro_function_name*)
Generate translation from transform short names 'korobov#x#' and 'sidi#' to C++ macros

Parameters **macro_function_name** – string; Name of the macro function to consider

Returns dict; A mapping between the transform short names and C++ macros

pySecDec.code_writer.template_parser.**parse_template_file**(*src*, *dest*, *replacements={}*)
Copy a file from *src* to *dest* replacing `%(...)` instructions in the standard python way.

Warning: If the file specified in *dest* exists, it is overwritten without prompt.

See also:

parse_template_tree()

Parameters

- **src** – str; The path to the template file.
- **dest** – str; The path to the destination file.
- **replacements** – dict; The replacements to be performed. The standard python replacement rules apply:

```
>>> '%(var)s = %(value)i' % dict(
...     var = 'my_variable',
...     value = 5)
'my_variable = 5'
```

pySecDec.code_writer.template_parser.**parse_template_tree**(*src*, *dest*, *replacements_in_files={}*, *filesystem_replacements={}*)
Copy a directory tree from *src* to *dest* using *parse_template_file()* for each file and replacing the filenames according to *filesystem_replacements*.

See also:

parse_template_file()

Parameters

- **src** – str; The path to the template directory.
- **dest** – str; The path to the destination directory.
- **replacements_in_files** – dict; The replacements to be performed in the files. The standard python replacement rules apply:

```
>>> '%(var)s = %(value)i' % dict(
...     var = 'my_variable',
...     value = 5)
'my_variable = 5'
```

- **filesystem_replacements** – dict; Renaming rules for the destination files. and directories. If a file or directory name in the source tree *src* matches a key in this dictionary, it is renamed to the corresponding value. If the value is `None`, the corresponding file is ignored.

pySecDec.code_writer.template_parser.**validate_pylink_qmc_transforms**(*pylink_qmc_transforms*)
Check if *pylink_qmc_transforms* are valid options and remove duplicates

Parameters **pylink_qmc_transforms** – list or None; Required qmc integral transforms, options are:

- korobov<i>x<j> for 1 <= i,j <= 6
- korobov<i> for 1 <= i <= 6 (same as korobov<i>x<i>)
- sidi<i> for 1 <= i <= 6

Returns Sorted set of pylink_qmc_transforms

5.9 Generated C++ Libraries

A C++ Library to numerically compute a given integral/ loop integral can be generated by the *sum_package()*, `loop_package()` functions. The *name* passed to the *make_package()* or *loop_package()* function will be used as the C++ namespace of the generated library. A program demonstrating the use of the C++ library is generated for each integral and written to `name/integrate_name.cpp`. Here we document the C++ library API.

See also:

C++ Interface

5.9.1 Amplitude/Sum libraries

New in version 1.5.

Generated by *sum_package()* in the folder `name`.

const unsigned long long **number_of_integrals**
The number of integrals in the library.

const unsigned int **number_of_amplitudes**
The number of amplitudes in the library.

const unsigned int **number_of_real_parameters**
The number of real parameters on which the integral depends.

const std::vector<std::string> **names_of_real_parameters**
An ordered vector of string representations of the names of the real parameters.

const unsigned int **number_of_complex_parameters**
The number of complex parameters on which the integral depends.

const std::vector<std::string> **names_of_complex_parameters**
An ordered vector of string representations of the names of the complex parameters.

const unsigned int **number_of_regulators**
The number of regulators on which the integral depends.

const std::vector<std::string> **names_of_regulators**
A vector of the names of the regulators.

const std::vector<int> **requested_orders**
A vector of the requested orders of each regulator used to generate the C++ library, i.e. the *requested_orders* parameter passed to *make_package()*, *loop_package()* or *sum_package()*.

typedef double **real_t**
The real type used by the library.

typedef std::complex<*real_t*> **complex_t**
The complex type used by the library.

const unsigned int maximal_number_of_integration_variables = 2;

type **integrand_return_t**
The return type of the integrand function. If the integral has complex parameters or uses contour deformation or if *enforce_complex* is set to `True` in the call to *make_package()* or *loop_package()* then *integrand_return_t* is *complex_t*. Otherwise *integrand_return_t* is *real_t*.

template<typename T> nested_series_t = secdecutil::Series<secdecutil::Series<...<T>>>
A potentially nested *secdecutil::Series* representing the series expansion in each of the regulators. If the integral depends on only one regulator (for example, a loop integral generated with *loop_package()*) this type will be a *secdecutil::Series*. For integrals that depend on multiple regulators then this will be a series of series representing the multivariate series. This type can be used to write code that can handle integrals depending on arbitrarily many regulators.

See also:

secdecutil::Series

template<typename T> using amplitudes_t = std::vector<nested_series_t<T>>
A vector of nested *secdecutil::Series* representing the amplitudes.

typedef secdecutil::IntegrandContainer<integrand_return_t, real_t const * const, real_t> integrand_t
The type of the integrands. Within the generated C++ library integrands are stored in a container along with the number of integration variables upon which they depend. These containers can be passed to an integrator for numerical integration.

type **cuda_integrand_t**
The type of a single integrand (sector) usable on a CUDA device (GPU). This container can be passed to an integrator for numerical integration.

See also:

secdecutil::IntegrandContainer, *secdecutil::Integrator*, and *secdecutil::integrators::Qmc*.

typedef integrand_t user_integrand_t
A convenience type of referring to either an *integrand_t* or *cuda_integrand_t* if the library was built with a CUDA compatible compiler.

typedef secdecutil::amplitude::Integral<integrand_return_t,real_t> integral_t
The type of the amplitude integral wrapper.

> **Warning:** The precise definition and usage of `secdecutil::amplitude::Integral` is likely to change in future versions of *pySecDec*.

typedef secdecutil::amplitude::WeightedIntegral<integral_t, integrand_return_t> weighted_integral_t
The type of the weighted integral. Weighted integrals consist of an integral, *I*, and the coefficient of the integral, *C*. A *WeightedIntegral* is interpreted as the product *C*I* and can be used to represent individual terms in an amplitude.

typedef std::vector<weighted_integral_t> sum_t
The type of a sum of weighted integrals.

template<template<typename...
> class container_t> using handler_t = secdecutil::amplitude::WeightedIntegralHandler<integrand_return_
real_t,integrand_return_t,container_t>
The type of the weighted integral handler. A *WeightedIntegralHandler* defines an algorithm for evaluating a sum of weighted integrals.

std::vector<nested_series_t<sum_t>> **make_amplitudes**(const std::vector<*real_t*> &real_parameters, const std::vector<*complex_t*> &complex_parameters, const std::string &lib_path, const integrator_t &integrator)
(without contour deformation)

std::vector<nested_series_t<sum_t>> **make_amplitudes**(const std::vector<*real_t*> &real_parameters, const std::vector<*complex_t*> &complex_parameters, const std::string &lib_path, const integrator_t &integrator, unsigned number_of_presamples = 100000, *real_t* deformation_parameters_maximum = 1., *real_t* deformation_parameters_minimum = 1.e-5, *real_t* deformation_parameters_decrease_factor = 0.9)
(with contour deformation)

Constructs and returns a vector of amplitudes ready to be passed to a *WeightedIntegralHandler* for evaluation. Each element of the vector contains an amplitude (weighted sum of integrals). The real and complex parameters are bound to the values passed in *real_parameters* and *complex_parameters*. The *lib_path* parameter is used to specify the path to the coefficients and individual integral libraries. The *integrator* parameter is used to specify which integrator should be used to evaluate the integrals. If enabled, contour deformation is controlled by the parameters *number_of_presamples*, *deformation_parameters_maximum*, *deformation_parameters_minimum*, *deformation_parameters_decrease_factor* which are documented in *pySecDec.integral_interface.IntegralLibrary*.

5.9.2 Integral libraries

Generated by *make_package()* in the folder `name` or by *sum_package()* in the folder `name/name_integral`.

typedef double **real_t**
The real type used by the library.

typedef std::complex<*real_t*> **complex_t**
The complex type used by the library.

type **integrand_return_t**
The return type of the integrand function. If the integral has complex parameters or uses contour deformation or if *enforce_complex* is set to `True` in the call to *make_package()* or *loop_package()* then *integrand_return_t* is *complex_t*. Otherwise *integrand_return_t* is *real_t*.

template<typename T> nested_series_t = secdecutil::Series<secdecutil::Series<...<T>>>
A potentially nested *secdecutil::Series* representing the series expansion in each of the regulators. If the integral depends on only one regulator (for example, a loop integral generated with *loop_package()*) this type will be a *secdecutil::Series*. For integrals that depend on multiple regulators then this will be a series of series representing the multivariate series. This type can be used to write code that can handle integrals depending on arbitrarily many regulators.

See also:

secdecutil::Series

typedef secdecutil::*IntegrandContainer*<*integrand_return_t*, *real_t* const*const> **integrand_t**
The type of the integrand. Within the generated C++ library integrands are stored in a container along with the number of integration variables upon which they depend. These containers can be passed to an integrator for numerical integration.

See also:

secdecutil::IntegrandContainer and *secdecutil::Integrator*.

type **cuda_integrand_t**
New in version 1.4.

The type of a single integrand (sector) usable on a CUDA device (GPU). This container can be passed to an integrator for numerical integration.

See also:

secdecutil::IntegrandContainer, *secdecutil::Integrator*, and *secdecutil::integrators::Qmc*.

type **cuda_together_integrand_t**
New in version 1.4.

The type of a sum of integrands (sectors) usable on a CUDA device (GPU). This container can be passed to an integrator for numerical integration.

See also:

secdecutil::IntegrandContainer, *secdecutil::Integrator*, and *secdecutil::integrators::Qmc*.

const unsigned long long **number_of_sectors**
The number of sectors generated by the sector decomposition.

Changed in version 1.3.1: Type was `unsigned int` in earlier versions of *pySecDec*.

const unsigned int **maximal_number_of_integration_variables**
The number of integration variables after primary decomposition. This provides an upper bound in the number of integration variables for all integrand functions. The actual number of integration variables may be lower for a given integrand.

const unsigned int **number_of_regulators**
The number of regulators on which the integral depends.

const unsigned int **number_of_real_parameters**
The number of real parameters on which the integral depends.

const std::vector<std::string> **names_of_real_parameters**
An ordered vector of string representations of the names of the real parameters.

const unsigned int **number_of_complex_parameters**
The number of complex parameters on which the integral depends.

const std::vector<std::string> **names_of_complex_parameters**
An ordered vector of string representations of the names of the complex parameters.

const std::vector<int> **lowest_orders**
A vector of the lowest order of each regulator which appears in the integral, not including the prefactor.

const std::vector<int> **highest_orders**
A vector of the highest order of each regulator which appears in the integral, not including the prefactor. This depends on the *requested_orders* and *prefactor*/*additional_prefactor* parameter passed to *make_package()* or *loop_package()*. In the case of *loop_package()* it also depends on the Γ-function prefactor of the integral which appears upon Feynman parametrization.

const std::vector<int> **lowest_prefactor_orders**
A vector of the lowest order of each regulator which appears in the prefactor of the integral.

const std::vector<int> **highest_prefactor_orders**
A vector of the highest order of each regulator which appears in the prefactor of the integral.

const std::vector<int> **requested_orders**
A vector of the requested orders of each regulator used to generate the C++ library, i.e. the *requested_orders* parameter passed to `make_package()` or `loop_package()`.

const std::vector<nested_series_t<sector_container_t>> &**get_sectors**()
Changed in version 1.3.1: The variable `sectors` has been replaced by this function.

A low level interface for obtaining the underlying integrand C++ functions.

> **Warning:** The precise definition and usage of `get_sectors()` is likely to change in future versions of *pySecDec*.

nested_series_t<*integrand_return_t*> **prefactor**(const std::vector<*real_t*> &real_parameters, const std::vector<*complex_t*> &complex_parameters)
The series expansion of the integral prefactor evaluated with the given parameters. If the library was generated using `make_package()` it will be equal to the *prefactor* passed to `make_package()`. If the library was generated with `loop_package()` it will be the product of the *additional_prefactor* passed to `loop_package()` and the Γ-function prefactor of the integral which appears upon Feynman parametrization.

const std::vector<std::vector<*real_t*>> **pole_structures**
A vector of the powers of the monomials that can be factored out of each sector of the polynomial during the decomposition.

Example: an integral depending on variables x and y may have two sectors, the first may have a monomial $x^{-1}y^{-2}$ factored out and the second may have a monomial x^{-1} factored out during the decomposition. The resulting *pole_structures* would read `{ {-1,-2}, {-1,0} }`. Poles of type x^{-1} are known as logarithmic poles, poles of type x^{-2} are known as linear poles.

std::vector<nested_series_t<*integrand_t*>> **make_integrands**(const std::vector<*real_t*> &real_parameters, const std::vector<*complex_t*> &complex_parameters)
(without contour deformation)

std::vector<nested_series_t<*cuda_integrand_t*>> **make_cuda_integrands**(const std::vector<*real_t*> &real_parameters, const std::vector<*complex_t*> &complex_parameters)
New in version 1.4.

(without contour deformation) (CUDA only)

std::vector<nested_series_t<*integrand_t*>> **make_integrands**(const std::vector<*real_t*> &real_parameters, const std::vector<*complex_t*> &complex_parameters, unsigned number_of_presamples = 100000, *real_t* deformation_parameters_maximum = 1., *real_t* deformation_parameters_minimum = 1.e-5, *real_t* deformation_parameters_decrease_factor = 0.9)
(with contour deformation)

std::vector<nested_series_t<*cuda_integrand_t*>> **make_cuda_integrands**(const std::vector<*real_t*> &real_parameters, const std::vector<*complex_t*> &complex_parameters, unsigned number_of_presamples = 100000, *real_t* deformation_parameters_maximum = 1., *real_t* deformation_parameters_minimum = 1.e-5, *real_t* deformation_parameters_decrease_factor = 0.9)

New in version 1.4.

(with contour deformation) (CUDA only)

Gives a vector containing the series expansions of individual sectors of the integrand after sector decomposition with the specified *real_parameters* and *complex_parameters* bound. Each element of the vector contains the series expansion of an individual sector. The series consists of instances of `integrand_t` (`cuda_integrand_t`) which contain the integrand functions and the number of integration variables upon which they depend. The real and complex parameters are bound to the values passed in *real_parameters* and *complex_parameters*. If enabled, contour deformation is controlled by the parameters *number_of_presamples*, *deformation_parameters_maximum*, *deformation_parameters_minimum*, *deformation_parameters_decrease_factor* which are documented in `pySecDec.integral_interface.IntegralLibrary`. In case of a sign check error (sign_check_error), manually set *number_of_presamples*, *deformation_parameters_maximum*, and *deformation_parameters_minimum*.

Passing the *integrand_t* to the `secdecutil::Integrator::integrate()` function of an instance of a particular `secdecutil::Integrator` will return the numerically evaluated integral. To integrate all orders of all sectors `secdecutil::deep_apply()` can be used.

Note: This is the recommended way to access the integrand functions.

See also:

C++ Interface, *Integrator Examples*, `pySecDec.integral_interface.IntegralLibrary`

5.10 Integral Interface

An interface to libraries generated by `pySecDec.code_writer.make_package()` or `pySecDec.loop_integral.loop_package()`.

class pySecDec.integral_interface.**CPPIntegrator**

Abstract base class for integrators to be used with an `IntegralLibrary`. This class holds a pointer to the c++ integrator and defines the destructor.

class pySecDec.integral_interface.**CQuad**(*integral_library*, *epsrel=0.01*, *epsabs=1e-07*, *n=100*, *verbose=False*, *zero_border=0.0*)

Wrapper for the cquad integrator defined in the gsl library.

Parameters **integral_library** – `IntegralLibrary`; The integral to be computed with this integrator.

The other options are defined in Section 4.6.1 and in the gsl manual.

class pySecDec.integral_interface.**CudaQmc**(*integral_library*, *transform='korobov3'*, *fitfunction='default'*, *generatingvectors='default'*, *epsrel=0.01*, *epsabs=1e-07*, *maxeval=4611686018427387903*, *errormode='default'*, *evaluateminn=0*, *minn=10000*, *minm=0*, *maxnperpackage=0*, *maxmperpackage=0*, *cputhreads=None*, *cudablocks=0*, *cudathreadsperblock=0*, *verbosity=0*, *seed=0*, *devices=[]*)

Wrapper for the Qmc integrator defined in the integrators library for GPU use.

Parameters

- **integral_library** – *IntegralLibrary*; The integral to be computed with this integrator.
- **errormode** – string; The *errormode* parameter of the Qmc, can be `"default"`, `"all"`, and `"largest"`. `"default"` takes the default from the Qmc library. See the Qmc docs for details on the other settings.
- **transform** – string; An integral transform related to the parameter P of the Qmc. The possible choices correspond to the integral transforms of the underlying Qmc implementation. Possible values are, `"none"`, `"baker"`, `sidi#`, `"korobov#"`, and `korobov#x#` where any `#` (the rank of the Korobov/Sidi transform) must be an integer between 1 and 6.
- **fitfunction** – string; An integral transform related to the parameter F of the Qmc. The possible choices correspond to the integral transforms of the underlying Qmc implementation. Possible values are `"default"`, `"none"`, `"polysingular"`.
- **generatingvectors** – string; The name of a set of generating vectors. The possible choices correspond to the available generating vectors of the underlying Qmc implementation. Possible values are `"default"`, `"cbcpt_dn1_100"`, `"cbcpt_dn2_6"`, `"cbcpt_cfftw1_6"`, and `"cbcpt_cfftw2_10"`.
- **cputhreads** – int; The number of CPU threads that should be used to evaluate the integrand function.

  The default is the number of logical CPUs allocated to the current process (that is, `len(os.sched_getaffinity(0))`) on platforms that expose this information (i.e. Linux+glibc), or `os.cpu_count()`.

  If GPUs are used, one additional CPU thread per device will be launched for communicating with the device. One can set ``cputhreads" to zero to disable CPU evaluation in this case.

See also:

The most important options are described in Section 4.6.2.

The other options are defined in the Qmc docs. If an argument is omitted then the default of the underlying Qmc implementation is used.

class pySecDec.integral_interface.**Cuhre**(*integral_library*, *epsrel=0.01*, *epsabs=1e-07*, *flags=0*, *mineval=10000*, *maxeval=4611686018427387903*, *zero_border=0.0*, *key=0*, *real_complex_together=False*)

Wrapper for the Cuhre integrator defined in the cuba library.

Parameters **integral_library** – *IntegralLibrary*; The integral to be computed with this integrator.

The other options are defined in Section 4.6.3 and in the cuba manual.

class pySecDec.integral_interface.**Divonne**(*integral_library*, *epsrel=0.01*, *epsabs=1e-07*, *flags=0*, *seed=0*, *mineval=10000*, *maxeval=4611686018427387903*, *zero_border=0.0*, *key1=2000*, *key2=1*, *key3=1*, *maxpass=4*, *border=0.0*, *maxchisq=1.0*, *mindeviation=0.15*, *real_complex_together=False*)

Wrapper for the Divonne integrator defined in the cuba library.

Parameters **integral_library** – *IntegralLibrary*; The integral to be computed with this integrator.

The other options are defined in Section 4.6.3 and in the cuba manual.

class pySecDec.integral_interface.**IntegralLibrary**(*shared_object_path*)

Interface to a c++ library produced by *make_package()* or *loop_package()*.

Parameters **shared_object_path** – str; The path to the file "<name>_pylink.so" that can be built by the command

```
$ make pylink
```

in the root directory of the c++ library.

Instances of this class can be called with the following arguments:

Parameters

- **real_parameters** – iterable of float; The real_parameters of the library.
- **complex_parameters** – iterable of complex; The complex parameters of the library.
- **together** – bool, optional; Whether to integrate the sum of all sectors or to integrate the sectors separately. Default: `True`.
- **number_of_presamples** – unsigned int, optional; The number of samples used for the contour optimization. A larger value here may resolve a sign check error (sign_check_error). This option is ignored if the integral library was created without deformation. Default: `100000`.
- **deformation_parameters_maximum** – float, optional; The maximal value the deformation parameters λ_i can obtain. Lower this value if you get a sign check error (sign_check_error). If `number_of_presamples=0`, all λ_i are set to this value. This option is ignored if the integral library was created without deformation. Default: `1.0`.
- **deformation_parameters_minimum** – float, optional; The minimal value the deformation parameters λ_i can obtain. Lower this value if you get a sign check error (sign_check_error). If `number_of_presamples=0`, all λ_i are set to this value. This option is ignored if the integral library was created without deformation. Default: `1e-5`.
- **deformation_parameters_decrease_factor** – float, optional; If the sign check with the optimized λ_i fails during the presampling stage, all λ_i are multiplied by this value until the sign check passes. We recommend to rather change `number_of_presamples`, `deformation_parameters_maximum`, and `deformation_parameters_minimum` in case of a sign check error. This option is ignored if the integral library was created without deformation. Default: `0.9`.
- **epsrel** – float, optional; The desired relative accuracy for the numerical evaluation of the weighted sum of the sectors. Default: epsrel of integrator (default 0.2).
- **epsabs** – float, optional; The desired absolute accuracy for the numerical evaluation of the weighted sum of the sectors. Default: epsabs of integrator (default 1e-7).

- **maxeval** – unsigned int, optional; The maximal number of integrand evaluations for each sector. Default: maxeval of integrator (default 2**62-1).
- **mineval** – unsigned int, optional; The minimal number of integrand evaluations for each sector. Default: mineval/minn of integrator (default 10000).
- **maxincreasefac** – float, optional; The maximum factor by which the number of integrand evaluations will be increased in a single refinement iteration. Default: `20`.
- **min_epsrel** – float, optional; The minimum relative accuracy required for each individual sector. Default: `0.2`
- **min_epsabs** – float, optional; The minimum absolute accuracy required for each individual sector. Default: `1.e-4`.
- **max_epsrel** – float, optional; The maximum relative accuracy assumed possible for each individual sector. Any sector known to this precision will not be refined further. Note: if this condition is met this means that the expected precision will not match the desired precision. Default: `1.e-14`.
- **max_epsabs** – float, optional; The maximum absolute accuracy assumed possible for each individual sector. Any sector known to this precision will not be refined further. Note: if this condition is met this means that the expected precision will not match the desired precision. Default: `1.e-20`.
- **min_decrease_factor** – float, optional; If the next refinement iteration is expected to make the total time taken for the code to run longer than `wall_clock_limit` then the number of points to be requested in the next iteration will be reduced by at least `min_decrease_factor`. Default: `0.9`.
- **decrease_to_percentage** – float, optional; If `remaining_time * decrease_to_percentage > time_for_next_iteration` then the number of points requested in the next refinement iteration will be reduced. Here: `remaining_time = wall_clock_limit - elapsed_time` and `time_for_next_iteration` is the estimated time required for the next refinement iteration. Note: if this condition is met this means that the expected precision will not match the desired precision. Default: `0.7`.
- **wall_clock_limit** – float, optional; If the current elapsed time has passed `wall_clock` limit and a refinement iteration finishes then a new refinement iteration will not be started. Instead, the code will return the current result and exit. Default: `DBL_MAX (1.7976931348623158e+308)`.
- **number_of_threads** – int, optional; The number of threads used to compute integrals concurrently. Note: The integrals themselves may also be computed with multiple threads irrespective of this option. Default: `0`.
- **reset_cuda_after** – int, optional; The cuda driver does not automatically remove unnecessary functions from the device memory such that the device may run out of memory after some time. This option controls after how many integrals `cudaDeviceReset()` is called to clear the memory. With the default `0`, `cudaDeviceReset()` is never called. This option is ignored if compiled without cuda. Default: `0 (never)`.
- **verbose** – bool, optional; Controls the verbosity of the output of the amplitude. Default: `False`.
- **errormode** – str, optional; Allowed values: `abs`, `all`, `largest`, `real`, `imag`. Defines how epsrel and epsabs should be applied to complex values. With the choice `largest`, the relative uncertainty is defined as `max( |Re(error)|, |Im(error)|)/max( |Re(result)|, |Im(result)|)`. Choosing `all` will apply epsrel and epsabs to both the real and imaginary part separately. Note: If either the real or imaginary part integrate to 0, the choices

`all`, `real` or `imag` might prevent the integration from stopping since the requested precision epsrel cannot be reached. Default: `abs`.

See also:

A more detailed description of these parameters and how they affect timing/precision is given in `chapter_cpp_amplitude`.

The call operator returns three strings: * The integral without its prefactor * The prefactor * The integral multiplied by the prefactor

The integrator can be configured by calling the member methods `use_Vegas()`, `use_Suave()`, `use_Divonne()`, `use_Cuhre()`, `use_CQuad()`, and `use_Qmc()`. The available options are listed in the documentation of *Vegas*, *Suave*, *Divonne*, *Cuhre*, *CQuad*, *Qmc* (*CudaQmc* for GPU version), respectively. *CQuad* can only be used for one dimensional integrals. A call to `use_CQuad()` configures the integrator to use *CQuad* if possible (1D) and the previously defined integrator otherwise. By default, *CQuad* (1D only) and *Vegas* are used with their default arguments. For details about the options, refer to the cuba and the gsl manual.

Further information about the library is stored in the member variable *info* of type `dict`.

class pySecDec.integral_interface.**MultiIntegrator**(*integral_library*, *low_dim_integrator*, *high_dim_integrator*, *critical_dim*)

New in version 1.3.1.

Wrapper for the *secdecutil::MultiIntegrator*.

Parameters

- **integral_library** – *IntegralLibrary*; The integral to be computed with this integrator.
- **low_dim_integrator** – *CPPIntegrator*; The integrator to be used if the integrand is lower dimensional than *critical_dim*.
- **high_dim_integrator** – *CPPIntegrator*; The integrator to be used if the integrand has dimension *critical_dim* or higher.
- **critical_dim** – integer; The dimension below which the *low_dimensional_integrator* is used.

Use this class to switch between integrators based on the dimension of the integrand when integrating the *integral_library*. For example, "*CQuad* for 1D and *Vegas* otherwise" is implemented as:

```
integral_library.integrator = MultiIntegrator(integral_library,CQuad(integral_
↪library),Vegas(integral_library),2)
```

MultiIntegrator can be nested to implement multiple critical dimensions. To use e.g. *CQuad* for 1D, *Cuhre* for 2D and 3D, and *Vegas* otherwise, do:

```
integral_library.integrator = MultiIntegrator(integral_library,CQuad(integral_
↪library),MultiIntegrator(integral_library,Cuhre(integral_library),Vegas(integral_
↪library),4),2)
```

Warning: The *integral_library* passed to the integrators must be the same for all of them. Furthermore, an integrator can only be used to integrate the *integral_library* it has beeen constructed with.

Warning: The *MultiIntegrator* cannot be used with *CudaQmc*.

class pySecDec.integral_interface.**Qmc**(*integral_library*, *transform='korobov3'*, *fitfunction='default'*, *generatingvectors='default'*, *epsrel=0.01*, *epsabs=1e-07*, *maxeval=4611686018427387903*, *errormode='default'*, *evaluateminn=0*, *minn=10000*, *minm=0*, *maxnperpackage=0*, *maxmperpackage=0*, *cputhreads=None*, *cudablocks=0*, *cudathreadsperblock=0*, *verbosity=0*, *seed=0*, *devices=[]*)

Wrapper for the Qmc integrator defined in the integrators library.

Parameters

- **integral_library** – *IntegralLibrary*; The integral to be computed with this integrator.
- **errormode** – string; The *errormode* parameter of the Qmc, can be `"default"`, `"all"`, and `"largest"`. `"default"` takes the default from the Qmc library. See the Qmc docs for details on the other settings.
- **transform** – string; An integral transform related to the parameter P of the Qmc. The possible choices correspond to the integral transforms of the underlying Qmc implementation. Possible values are, `"none"`, `"baker"`, `sidi#`, `"korobov#"`, and `korobov#x#` where any `#` (the rank of the Korobov/Sidi transform) must be an integer between 1 and 6.
- **fitfunction** – string; An integral transform related to the parameter F of the Qmc. The possible choices correspond to the integral transforms of the underlying Qmc implementation. Possible values are `"default"`, `"none"`, `"polysingular"`.
- **generatingvectors** – string; The name of a set of generating vectors. The possible choices correspond to the available generating vectors of the underlying Qmc implementation. Possible values are `"default"`, `"cbcpt_dn1_100"`, `"cbcpt_dn2_6"`, `"cbcpt_cfftw1_6"`, and `"cbcpt_cfftw2_10"`.

  The `"default"` value will use all available generating vectors suitable for the highest dimension integral appearing in the library.
- **cputhreads** – int; The number of CPU threads that should be used to evaluate the integrand function.

  The default is the number of logical CPUs allocated to the current process (that is, `len(os.sched_getaffinity(0))`) on platforms that expose this information (i.e. Linux+glibc), or `os.cpu_count()`.

  If GPUs are used, one additional CPU thread per device will be launched for communicating with the device. One can set ``cputhreads" to zero to disable CPU evaluation in this case.

See also:

The most important options are described in Section 4.6.2.

The other options are defined in the Qmc docs. If an argument is omitted then the default of the underlying Qmc implementation is used.

class pySecDec.integral_interface.**Suave**(*integral_library*, *epsrel=0.01*, *epsabs=1e-07*, *flags=0*, *seed=0*, *mineval=10000*, *maxeval=4611686018427387903*, *zero_border=0.0*, *nnew=1000*, *nmin=10*, *flatness=25.0*, *real_complex_together=False*)

Wrapper for the Suave integrator defined in the cuba library.

Parameters **integral_library** – *IntegralLibrary*; The integral to be computed with this integrator.

The other options are defined in Section 4.6.3 and in the cuba manual.

class pySecDec.integral_interface.**Vegas**(*integral_library*, *epsrel=0.01*, *epsabs=1e-07*, *flags=0*, *seed=0*, *mineval=10000*, *maxeval=4611686018427387903*, *zero_border=0.0*, *nstart=10000*, *nincrease=5000*, *nbatch=1000*, *real_complex_together=False*)

Wrapper for the Vegas integrator defined in the cuba library.

Parameters **integral_library** – *IntegralLibrary*; The integral to be computed with this integrator.

The other options are defined in Section 4.6.3 and in the cuba manual.

pySecDec.integral_interface.**series_to_ginac**(*series*)

Convert a textual representation of a series into GiNaC format.

Parameters **series** (*str*) – Any of the series obtained by calling an *IntegralLibrary* object.

Returns

Two strings: the series of mean values, and the series of standard deviations. The format of each returned value may look like this:

```
(0+0.012665*I)/eps + (0+0.028632*I) + Order(eps)
```

pySecDec.integral_interface.**series_to_maple**(*series*)

Convert a textual representation of a series into Maple format.

Parameters **series** (*str*) – Any of the series obtained by calling an *IntegralLibrary* object.

Returns

Two strings: the series of mean values, and the series of standard deviations. The format of each returned value may look like this:

```
(0+0.012665*I)/eps + (0+0.028632*I) + O(eps)
```

pySecDec.integral_interface.**series_to_mathematica**(*series*)

Convert a textual representation of a series into Mathematica format.

Parameters **series** (*str*) – Any of the series obtained by calling an *IntegralLibrary* object.

Returns

Two strings: the series of mean values, and the series of standard deviations. The format of each returned value may look like this:

```
(0+0.012665*I)/eps + (0+0.028632*I) + O[eps]
```

pySecDec.integral_interface.**series_to_sympy**(*series*)

Convert a textual representation of a series into SymPy format.

Parameters **series** (*str*) – Any of the series obtained by calling an *IntegralLibrary* object.

Returns

Two strings: the series of mean values, and the series of standard deviations. The format of each returned value may look like this:

```
(0+0.012665*I)/eps + (0+0.028632*I) + O(eps)
```

5.11 Miscellaneous

Collection of general-purpose helper functions.

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):
```

(continues on next page)

(continued from previous page)

```
        result = 0
        for i in range(1, self.N + 1):
            result += i
        return result
```

pySecDec.misc.**chunks**(*lst*, *n*)
Yield successive n-sized chunks from lst.

Parameters

- **lst** – list; The list from which to produce chunks.
- **n** – integer; The size of the chunks to produce.

Returns A list of at most length n.

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.**flatten**(*polynomial*, *depth=inf*)
Convert nested polynomials; i.e. polynomials that have polynomials in their coefficients to one single polynomial.

Parameters

- **polynomial** – *pySecDec.algebra.Polynomial*; The polynomial to "flatten".
- **depth** – integer; The maximum number of recursion steps. If not provided, stop if the coefficient is not a *pySecDec.algebra.Polynomial*.

pySecDec.misc.**lowest_order**(*expression*, *variable*)
Find the lowest order of *expression*'s series expansion in *variable*.

Example:

```
>>> from pySecDec.misc import lowest_order
>>> lowest_order('exp(eps)', 'eps')
0
>>> lowest_order('gamma(eps)', 'eps')
-1
```

Parameters

- **expression** – string or sympy expression; The expression to compute the lowest expansion order of.
- **variable** – string or sympy expression; The variable in which to expand.

pySecDec.misc.**make_cpp_list**(*python_list*)
Convert a python list to a string to be used in c++ initializer list.

Example: `['a', 'b', 'c'] --> '"a","b","c"'`

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.**parallel_det**(*M*, *pool*)

Calculate the determinant of a matrix in parallel.

Parameters

- **M** – a square-matrix-like array;
- **pool** – `multiprocessing.Pool`; The pool to be used.

Example:

```
>>> from pySecDec.misc import parallel_det
>>> from multiprocessing import Pool
>>> from sympy import sympify
>>> M = [['m11','m12','m13','m14'],
...      ['m21','m22','m23','m24'],
...      ['m31','m32','m33','m34'],
...      ['m41','m42','m43','m44']]
>>> M = sympify(M)
>>> parallel_det(M, Pool(2)) # 2 processes
m11*(m22*(m33*m44 - m34*m43) - m23*(m32*m44 - m34*m42) + m24*(m32*m43 - m33*m42)) -␣
↪m12*(m21*(m33*m44 - m34*m43) - m23*(m31*m44 - m34*m41) + m24*(m31*m43 - m33*m41))␣
↪+ m13*(m21*(m32*m44 - m34*m42) - m22*(m31*m44 - m34*m41) + m24*(m31*m42 -␣
↪m32*m41)) - m14*(m21*(m32*m43 - m33*m42) - m22*(m31*m43 - m33*m41) + m23*(m31*m42␣
↪- m32*m41))
```

pySecDec.misc.**powerset**(*iterable*, *min_length=0*, *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.
- **min_length** – integer, optional; Only generate sets with minimal given length. Default: `0`.
- **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.**rangecomb**(*low*, *high*)

Return an iterator over the occuring orders in a multivariate series expansion between *low* and *high*.

Parameters

- **low** – vector-like array; The lowest orders.
- **high** – vector-like array; The highest orders.

Example:

```
>>> from pySecDec.misc import rangecomb
>>> all_orders = rangecomb([-1,-2], [0,0])
>>> list(all_orders)
[(-1, -2), (-1, -1), (-1, 0), (0, -2), (0, -1), (0, 0)]
```

pySecDec.misc.**rec_subs**(*expr*, *repl*, *n=200*)
Substitute repl in expr and expand until expr stops changing or depth n is reached.

Parameters

- **expr** – sympy expression; Expression to which the replacement rules are applied.
- **repl** – list; List of replacement rules.
- **n** – integer; Maximal number of repl applications.

Returns Expression after substitutions.

pySecDec.misc.**sympify_expression**(*a*)
A helper function for converting objects to sympy expressions.

First try to convert object to a sympy expression, if this fails, then try to convert str(object) to a sympy expression

Parameters **a** –

The object to be converted to a sympy expression.

Returns

A sympy expression representing the object.

pySecDec.misc.**sympify_symbols**(*iterable*, *error_message*, *allow_number=False*)
sympify each item in *iterable* and assert that it is a *symbol*.

This module provides a make_package like interface to code_writer.sum_package `make_sum_package()`.

pySecDec.make_package.**make_package**(*name*, *integration_variables*, *regulators*, *requested_orders*, *polynomials_to_decompose*, *polynomial_names=[]*, *other_polynomials=[]*, *prefactor=1*, *remainder_expression=1*, *functions=[]*, *real_parameters=[]*, *complex_parameters=[]*, *form_optimization_level=2*, *form_work_space='50M'*, *form_memory_use=None*, *form_threads=2*, *form_insertion_depth=5*, *contour_deformation_polynomial=None*, *positive_polynomials=[]*, *decomposition_method='iterative_no_primary'*, *normaliz_executable='normaliz'*, *enforce_complex=False*, *split=False*, *ibp_power_goal=- 1*, *use_iterative_sort=True*, *use_light_Pak=True*, *use_dreadnaut=False*, *use_Pak=True*, *processes=None*, *form_executable=None*, *pylink_qmc_transforms=['korobov3x3']*)
Decompose, subtract and expand an expression. Return it as c++ package.

See also:

In order to decompose a loop integral, use the function *pySecDec.loop_integral.loop_package()*.

See also:

The generated library is described in *Generated C++ Libraries*.

See also:

pySecDec.code_writer.make_package()

Parameters

- **name** – string; The name of the c++ namepace and the output directory.
- **integration_variables** – iterable of strings or sympy symbols; The variables that are to be integrated. The intgration region depends on the chosen *decomposition_method*.
- **regulators** – iterable of strings or sympy symbols; The UV/IR regulators of the integral.
- **requested_orders** – iterable of integers; Compute the expansion in the regulators to these orders.
- **polynomials_to_decompose** – iterable of strings or sympy expressions or *pySecDec.algebra.ExponentiatedPolynomial* or *pySecDec.algebra.Polynomial*; The polynomials to be decomposed.
- **polynomial_names** – iterable of strings; Assign symbols for the *polynomials_to_decompose*. These can be referenced in the *other_polynomials*; see *other_polynomials* for details.
- **other_polynomials** – iterable of strings or sympy expressions or *pySecDec.algebra.ExponentiatedPolynomial* or *pySecDec.algebra.Polynomial*; Additional polynomials where no decomposition is attempted. The symbols defined in *polynomial_names* can be used to reference the *polynomials_to_decompose*. This is particularly useful when computing loop integrals where the "numerator" can depend on the first and second Symanzik polynomials.

  Example (1-loop bubble with numerator):

```
>>> polynomials_to_decompose = ["(x0 + x1)**(2*eps - 4)",
...                             "(-p**2*x0*x1)**(-eps))"]
>>> polynomial_names = ["U", "F"]
>>> other_polynomials = ["""   (eps - 1)*s*U**2
...                          + (eps - 2)*F
...                          - (eps - 1)*2*s*x0*U
...                          + (eps - 1)*s*x0**2"""]
```

  See also:

  pySecDec.loop_integral

  Note that the *polynomial_names* refer to the *polynomials_to_decompose* **without** their exponents.

- **prefactor** – string or sympy expression, optional; A factor that does not depend on the integration variables.
- **remainder_expression** – string or sympy expression or `pySecDec.algebra._Expression`, optional; An additional factor.

  Dummy function must be provided with all arguments, e.g. `remainder_expression='exp(eps)*f(x0,x1)'`. In addition, all dummy function must be listed in *functions*.
- **functions** – iterable of strings or sympy symbols, optional; Function symbols occuring in *remainder_expression*, e.g.``['f']``.

Note: Only user-defined functions that are provided as c++-callable code should be mentioned here. Listing basic mathematical functions (e.g. `log`, `pow`, `exp`, `sqrt`, ...) is not required and considered an error to avoid name conflicts.

Note: The power function *pow* and the logarithm *log* use the nonstandard continuation with an infinitesimal negative imaginary part on the negative real axis (e.g. `log(-1) = -i*pi`).

- **real_parameters** – iterable of strings or sympy symbols, optional; Symbols to be interpreted as real variables.
- **complex_parameters** – iterable of strings or sympy symbols, optional; Symbols to be interpreted as complex variables.
- **form_optimization_level** – integer out of the interval [0,4], optional; The optimization level to be used in FORM. Default: `2`.
- **form_work_space** – string, optional; The FORM WorkSpace. Default: `'500M'`.

  Setting this to smaller values will reduce FORM memory usage (without affecting performance), but each problem has some minimum value below which FORM will refuse to work: it will fail with error message indicating that larger WorkSpace is needed, at which point WorkSpace will be adjusted and FORM will be re-run.
- **form_memory_use** – string, optional; The target FORM memory usage. When specified, *form.set* parameters will be adjusted so that FORM uses at most approximately this much resident memory.

  The minimum is approximately to 600M + 350M per worker thread if `form_work_space` is left at `'50M'`. if form_work_space is increased to `'500M'`, then the minimum is 2.5G + 2.5G per worker thread. Default: `None`, meaning use the default FORM values.
- **form_threads** – integer, optional; Number of threads (T)FORM will use. Default: `2`.
- **form_insertion_depth** – nonnegative integer, optional; How deep FORM should try to resolve nested function calls. Default: `5`.
- **contour_deformation_polynomial** – string or sympy symbol, optional; The name of the polynomial in *polynomial_names* that is to be continued to the complex plane according to a $-i\delta$ prescription. For loop integrals, this is the second Symanzik polynomial `F`. If not provided, no code for contour deformation is created.
- **positive_polynomials** – iterable of strings or sympy symbols, optional; The names of the polynomials in *polynomial_names* that should always have a positive real part. For loop integrals, this applies to the first Symanzik polynomial `U`. If not provided, no polynomial is checked for positiveness. If *contour_deformation_polynomial* is `None`, this parameter is ignored.
- **decomposition_method** – string, optional; The strategy to decompose the polynomials. The following strategies are available:
  - 'iterative_no_primary' (default): integration region $[0,1]^N$.
  - 'geometric_no_primary': integration region $[0,1]^N$.
  - 'geometric_infinity_no_primary': integration region $[0,\infty]^N$.
  - 'iterative': primary decomposition followed by integration over $[0,1]^{N-1}$.
  - 'geometric': x_N is set to one followed by integration over $[0,\infty]^{N-1}$.
  - 'geometric_ku': primary decomposition followed by integration over $[0,1]^{N-1}$.

  'iterative', 'geometric', and 'geometric_ku' are only valid for loop integrals. An end user should use 'iterative_no_primary', 'geometric_no_primary', or 'geomet-

ric_infinity_no_primary' here. In order to compute loop integrals, please use the function *pySecDec.loop_integral.loop_package()*.

- **normaliz_executable** – string, optional; The command to run *normaliz*. *normaliz* is only required if *decomposition_method* starts with 'geometric'. Default: 'normaliz'
- **enforce_complex** – bool, optional; Whether or not the generated integrand functions should have a complex return type even though they might be purely real. The return type of the integrands is automatically complex if *contour_deformation* is `True` or if there are *complex_parameters*. In other cases, the calculation can typically be kept purely real. Most commonly, this flag is needed if `log(<negative real>)` occurs in one of the integrand functions. However, *pySecDec* will suggest setting this flag to `True` in that case. Default: `False`
- **split** – bool or integer, optional; Whether or not to split the integration domain in order to map singularities from 1 to 0. Set this option to `True` if you have singularties when one or more integration variables are one. If an integer is passed, that integer is used as seed to generate the splitting point. Default: `False`
- **ibp_power_goal** – number or iterable of number, optional; The *power_goal* that is forwarded to *integrate_by_parts()*.

  This option controls how the subtraction terms are generated. Setting it to `-numpy.inf` disables *integrate_by_parts()*, while `0` disables *integrate_pole_part()*.

  See also:

  To generate the subtraction terms, this function first calls *integrate_by_parts()* for each integration variable with the give *ibp_power_goal*. Then *integrate_pole_part()* is called.

  Default: `-1`
- **use_iterative_sort** – bool; Whether or not to use *squash_symmetry_redundant_sectors_sort()* with *iterative_sort()* to find sector symmetries. Default: `True`
- **use_light_Pak** – bool; Whether or not to use *squash_symmetry_redundant_sectors_sort()* with *light_Pak_sort()* to find sector symmetries. Default: `True`
- **use_dreadnaut** – bool or string, optional; Whether or not to use *squash_symmetry_redundant_sectors_dreadnaut()* to find sector symmetries. If given a string, interpret that string as the command line executable *dreadnaut*. If `True`, try `$SECDEC_CONTRIB/bin/dreadnaut` and, if the environment variable `$SECDEC_CONTRIB` is not set, `dreadnaut`. Default: `False`
- **use_Pak** – bool; Whether or not to use *squash_symmetry_redundant_sectors_sort()* with *Pak_sort()* to find sector symmetries. Default: `True`
- **processes** – integer or None, optional; The maximal number of processes to be used. If `None`, the number of CPUs `multiprocessing.cpu_count()` is used. *New in version 1.3*. Default: `None`
- **form_executable** – string or None, optional; The path to the form exectuable. The argument is passed to *Coefficient.process()*. If `None`, then either `$FORM`, `$SECDEC_CONTRIB/bin/form`, or just `form` is used, depending on which environment variable is set. Default: `None`.
- **pylink_qmc_transforms** – list or None, optional; Required qmc integral transforms, options are:

  – korobov<i>x<j> for 1 <= i,j <= 6

- `korobov<i>` for 1 <= i <= 6 (same as `korobov<i>x<i>`)
- `sidi<i>` for 1 <= i <= 6

New in version 1.5. Default: `['korobov3x3']`

5.12 Expansion by Regions

Routines to perform an expansion by regions, see e.g. [PS11].

`pySecDec.make_regions.`**`apply_region`**(*polynomials*, *region_vector*, *expansion_parameter_index*)

Apply the *region_vector* to the input *polynomials*.

Note: Redefines the *expansion_parameter* as $\rho \to \rho^n$, where n is given by the *region_vector*.

Note: *apply_region* modifies the input *polynomials*.

Parameters

- **`polynomials`** – iterable of polynomials; Polynomials to be computed in different regions.
- **`region_vector`** – vector-like array; Region specified by the power of the *expansion_parameter* in the rescaled variables. The region vectors have to be specified in the same order as the symbols are specified in the polynomials. E.g. if symbols are specified as ['x0','x1','rho'] and want rescaling x0 -> rho^i * x0, x1 -> rho^k * x1 and rho -> rho^n, then the region vector needs to be [i,k,n]
- **`expansion_parameter_index`** – integer; Index of the expansion parameter in the list of symbols.

`pySecDec.make_regions.`**`derive_prod`**(*poly_list*, *numerator*, *index*, *polynomial_name_indices*)

Calculates the derivative of a product of polynomials using

$$\frac{\partial}{\partial x_i}\left(\prod_j P_j^{\alpha_j} N\right) = \prod_j P_j^{\alpha_j - 1} N'$$

where N' is given by

$$N' = \left(\sum_j N \alpha_j \frac{\partial P_j}{\partial x_i} \prod_{k \neq j} P_k\right) + \left(\prod_l P_l\right)\left[\left(\sum_k \frac{\partial N}{\partial P_k}\frac{\partial P_k}{\partial x_i}\right) + \frac{\partial N}{\partial x_i}\right].$$

Parameters

- **`poly_list`** – list of *`ExponentiatedPolynomial`*; The exponentiated polynomials that should be differentiated. They need to be defined in terms of the symbols `x0,x1,x2,..` and `p0,p1,p2..` where `p0,p1,p2..` are the bases of the exponentiated polynomials.
- **`numerator`** – *`Polynomial`*; The numerator also defined as an exponentiated polynomial with `symbols = [x0,x1,...,p0,p1,...]`.
- **`index`** – integer; Index of variable with respect to which the derivative is taken.
- **`polynomial_name_indices`** – iterable; Indices of polynomial names in poly_symbols.

pySecDec.make_regions.**expand_region**(*poly_list*, *numerator*, *index*, *order*, *polynomial_name_indices*)

Expands the product of the polynomials in *poly_list* and the numerator with respect to the variable whose *index* is given to a desired order specified by *order*.

Parameters

- **poly_list** – list of *ExponentiatedPolynomial*; The exponentiated polynomials that should be expanded. They need to be defined in terms of the symbols `x0,x1,x2,..` and `p0,p1,p2..` where `p0,p1,p2..` are the bases of the exponentiated polynomials.
- **numerator** – *Polynomial*; The numerator also defined as an exponentiated polynomial with `symbols = [x0,x1,...,p0,p1,...]`.
- **index** – integer; Index of variable with respect to which the polynomials are expanded.
- **order** – integer; Desired order of expansion.
- **polynomial_name_indices** – list of int; Indices of polynomials in the symbols of the input polynomials.

pySecDec.make_regions.**find_regions**(*exp_param_index*, *polynomial*, *indices=None*, *normaliz='normaliz'*, *workdir='normaliz_tmp'*)

Find regions for the expansion by regions as described in [PS11].

Note: This function calls the command line executable of *normaliz* [BIR]. See *The Geomethod and Normaliz* for installation and a list of tested versions.

Parameters

- **exp_param_index** – int; The index of the expansion parameter in the expolist.
- **polynomials** – an instance of *Polynomial*, for which to calculate the regions for.
- **indices** – list of integers or None; The indices of the parameters to be included in the asymptotic expansion. This should include all Feynman parameters (integration variables) and the expansion parameter. By default (`indices=None`), all parameters are considered.
- **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.make_regions.**make_regions**(*name*, *integration_variables*, *regulators*, *requested_orders*, *smallness_parameter*, *polynomials_to_decompose*, *expansion_by_regions_order=0*, *real_parameters=[]*, *complex_parameters=[]*, *normaliz='normaliz'*, *polytope_from_sum_of=None*, ***make_package_args*)

Applies the expansion by regions method (see e.g. [PS11]) to a list of polynomials.

Parameters

- **name** – string; The name of the c++ namepace and the output directory.
- **integration_variables** – iterable of strings or sympy symbols; The variables that are to be integrated from 0 to 1.

- **regulators** – iterable of strings or sympy symbols; The regulators of the integral.
- **requested_orders** – iterable of integers; Compute the expansion in the regulators to these orders.
- **smallness_parameter** – string or sympy symbol; The symbol of the variable in which the expression is expanded.
- **polynomials_to_decompose** – iterable of strings or sympy expressions or *pySecDec.algebra.ExponentiatedPolynomial* or *pySecDec.algebra.Polynomial*; The polynomials to be decomposed.
- **expansion_by_regions_order** – integer; The order up to which the expression is expanded in the *smallness_parameter*. Default: 0
- **real_parameters** – iterable of strings or sympy symbols, optional; Symbols to be interpreted as real variables.
- **complex_parameters** – iterable of strings or sympy symbols, optional; Symbols to be interpreted as complex variables.
- **normaliz** – string; The shell command to run *normaliz*. Default: 'normaliz'
- **polytope_from_sum_of** – iterable of integers; If this value is None, the product of all the polynomials *polynomials_to_decompose* is used to determine the Newton polytope and the normal vectors to it. Otherwise, the sum of the polynomials with the indices given by this parameter are used.
- **make_package_args** – The arguments to be forwarded to *pySecDec.code_writer.make_package()*.

CHAPTER

SIX

FREQUENTLY ASKED QUESTIONS

6.1 How can I adjust the integrator parameters?

If the python interface is used for the numerical integration, i.e. a python script like `examples/integrate_box1L.py`, the integration parameters can be specified in the argument list of the integrator call. For example, using Vegas as integrator:

```
box1L.use_Vegas(flags=2, epsrel=1e-3, epsabs=1e-12, nstart=5000, nincrease=10000,␣
↪maxeval=10000000, real_complex_together=True)
```

Or, using Divonne as integrator:

```
box1L.use_Divonne(flags=2, epsrel=1e-3, epsabs=1e-12, maxeval=10000000, border=1e-8,␣
↪real complex together=True)
```

The parameter real complex together tells the integrator to integrate real and imaginary parts simultaneously. A complete list of possible options for the integrators can be found in *integral_interface*.

If the C++ interface is used, the options can be specified as fields of the integrator. For example, after running `examples/generate_box1L.py`, in the file `examples/box1L/integrate_box1L.cpp`, you can modify the corresponding block to e.g.:

```
// Integrate
secdecutil::cuba::Vegas<box1L::integrand_return_t> integrator;
integrator.flags = 2; // verbose output --> see cuba manual
integrator.epsrel = 1e-2;
integrator.epsabs = 1e-12;
integrator.nstart = 5000;
integrator.nincrease = 10000;
integrator.maxeval = 10000000;
integrator.together = true;
```

In order to set the Divonne integrator with the same parameters as above, do:

```
// Integrate
secdecutil::cuba::Divonne<box1L::integrand_return_t> integrator;
integrator.flags = 2; // verbose output --> see cuba manual
integrator.epsrel = 1e-2;
integrator.epsabs = 1e-12;
integrator.maxeval = 10000000;
integrator.border = 1e-8;
integrator.together = true;
```

More information about the C++ integrator class can be found in Section 4.6.

6.2 How can I request a higher numerical accuracy?

The integrator stops if any of the following conditions is fulfilled: (1) `epsrel` is reached, (2) `epsabs` is reached, (3) `maxeval` is reached. Therefore, setting these parameters accordingly will cause the integrator to make more iterations and reach a more accurate result.

6.3 What can I do if the integration takes very long?

For most integrals, the best performance will be achieved using the QMC integrator and we recommend switching to it, if not already used. If changing the integrator doesn't improve the runtime, it is possible that the integrator parameters should be adjusted, as described in the previous sections. In particular for integrals with spurious poles, the parameter `epsabs` should be increased, since it is the only relevant stopping criterion in this case, besides `maxeval`.

6.4 How can I tune the contour deformation parameters?

You can specify the parameters in the argument of the integral call in the python script for the integration, see e.g. line 12 of `examples/integrate box1L.py`:

```
str_integral_without_prefactor, str_prefactor, str_integral_with_prefactor=box1L(real_
↪parameters=[4.,-0.75,1.25,1.],number_of_presamples=10**6,deformation_parameters_
↪maximum=0.5)
```

This sets the number of presampling points to `10**6` (default: `10**5`) and the maximum value for the contour deformation parameter `deformation_parameters_maximum` to `0.5` (default: `1`). The user should make sure that deformation parameters maximum is always larger than deformation_parameters_minimum (default: `1e-5`). These parameters are described in *IntegralLibrary*.

6.5 What can I do if the program stops with an error message containing *sign_check_error*?

This error occurs if the contour deformation leads to a wrong sign of the Feynman $i\delta$ prescription, usually due to the fact that the deformation parameter λ is too large. If this error is encountered the program will automatically reduce λ and re-attempt integration. If the code continues after the error and eventually returns a result then it successfully adjusted the contour and the error can be ignored. To avoid this error in the first place choose a larger value for `number_of_presamples` and a smaller value (e.g. `0.5`) for `deformation_parameters_maximum` (see item above). If that does not help, you can try `0.1` instead of `0.5` for `deformation_parameters_maximum`. The relevant parameters are described in *IntegralLibrary*.

If the code fails to find a contour it may display the error message `All deformation parameters at minimum already, integral still fails` and stop. In this case try reducing `deformation_parameters_maximum` (default: `1e-5`) to a smaller number. If the code still fails to find a valid contour it may be that your integral has an unavoidable end-point singularity or other numerical problems. Often this error is encountered when the `real_parameters` and/or `complex_parameters` are very large/small or if some of the parameters differ from each other by orders of magnitude. If all of the `real_parameters` or `complex_parameters` are of a similar size (but not $\mathcal{O}(1)$) then dividing each parameter by e.g. the largest parameter (such that all parameters are $\mathcal{O}(1)$) can help to avoid a situation where

extremely small deformation parameters are required to obtain a valid contour. It may then be possible to restore the desired result using dimensional analysis (i.e. multiplying the result by some power of the largest parameter).

If you still encounter an error after following these suggestions, please open an issue.

6.6 What does *additional_prefactor* mean exactly?

We should first point out that the conventions for additional prefactors defined by the user have been changed between *SecDec 3* and *pySecDec*. The prefactor specified by the user will now be *included* in the numerical result.

To make clear what is meant by "additional", we repeat our conventions for Feynman integrals here.

A scalar Feynman graph G in D dimensions at L loops with N propagators, where the propagators can have arbitrary, not necessarily integer powers ν_j, has the following representation in momentum space:

$$G = \int \prod_{l=1}^{L} \mathrm{d}^D \kappa_l \; \frac{1}{\prod\limits_{j=1}^{N} P_j^{\nu_j}(\{k\},\{p\},m_j^2)},$$

$$\mathrm{d}^D \kappa_l = \frac{\mu^{4-D}}{i\pi^{\frac{D}{2}}} \mathrm{d}^D k_l \;,\; P_j(\{k\},\{p\},m_j^2) = (q_j^2 - m_j^2 + i\delta) \;,$$

where the q_j are linear combinations of external momenta p_i and loop momenta k_l.

Introducing Feynman parameters leads to:

$$G = (-1)^{N_\nu} \frac{\Gamma(N_\nu - LD/2)}{\prod_{j=1}^{N} \Gamma(\nu_j)} \int\limits_0^\infty \prod_{j=1}^{N} dx_j \; x_j^{\nu_j - 1} \, \delta(1 - \sum_{l=1}^{N} x_l) \, \frac{\mathcal{U}^{N_\nu - (L+1)D/2}}{\mathcal{F}^{N_\nu - LD/2}}$$

The prefactor $(-1)^{N_\nu} \, \Gamma(N_\nu - LD/2) / \prod_{j=1}^{N} \Gamma(\nu_j)$ coming from the Feynman parametrisation will always be included in the numerical result, corresponding to *additional_prefactor=1* (default), i.e. the program will return the numerical value for G. If the user defines *additional_prefactor='gamma(3-2*eps)'*, this prefactor will be expanded in ϵ and included in the numerical result returned by *pySecDec*, in addition to the one coming from the Feynman parametrisation.

For general polynomials not related to loop integrals, i.e. in `make_package`, the prefactor provided by the user is the only prefactor, as there is no prefactor coming from a Feynman parametrisation in this case. This is the reason why in *make_package* the keyword for the prefactor defined by the user is `prefactor`, while in *loop_package* it is `additional_prefactor`.

6.7 What can I do if I get *nan*?

This means that the integral does not converge which can have several reasons. When Divonne is used as an integrator, it is important to use a non-zero value for border, e.g. `border=1e-8`. Vegas is in general the most robust integrator. When using Vegas, try to increase the values for `nstart` and `nincrease`, for example `nstart=100000` (default: `10000`) and `nincrease=50000` (default: `5000`).

If the integral is non-Euclidean, make sure that *contour_deformation=True* is set. Another reason for getting *nan* can be that the integral has singularities at $x_i = 1$ and therefore needs usage of the `split` option, see item below.

6.8 What can I use as numerator of a loop integral?

The numerator must be a sum of products of numbers, scalar products (e.g. `p1(mu)*k1(mu)*p1(nu)*k2(nu)` and/or symbols (e.g. `m`). The numerator can also be an inverse propagator. In addition, the numerator must be finite in the limit $\epsilon \to 0$. The default numerator is 1.

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)
```

More details can be found in *LoopIntegralFromPropagators*.

6.9 How can I integrate just one coefficient of a particular order in the regulator?

You can pick a certain order in the C++ interface (see *C++ Interface (advanced)*). To integrate only one order, for example the finite part, change the line:

```
const box1L::nested_series_t<secdecutil::UncorrelatedDeviation<box1L::integrand_return_t>
↪> result_all = secdecutil::deep_apply( all_sectors, integrator.integrate );
```

to:

```
int order = 0; // compute finite part only
const secdecutil::UncorrelatedDeviation<box1L::integrand_return_t> result_order =␣
↪secdecutil::deep_apply(all_sectors.at(order), integrator.integrate );
```

where `box1L` is to be replaced by the name of your integral. In addition, you should change the lines:

```
std::cout << "-- integral without prefactor -- " << std::endl;
std::cout << result_all << std::endl << std::endl;
```

to:

```
std::cout << "-- integral without prefactor -- " << std::endl;
std::cout << result_order << std::endl << std::endl;
```

and remove the lines:

```
std::cout << "-- prefactor -- " << std::endl;
const box1L::nested_series_t<box1L::integrand_return_t> prefactor =␣
↪box1L::prefactor(real_parameters, complex_parameters);
std::cout << prefactor << std::endl << std::endl;

std::cout << "-- full result (prefactor*integral) -- " << std::endl;
std::cout << prefactor*result_all << std::endl;
```

because the expansion of the prefactor will in general mix with the pole coefficients and thus affect the finite part. We should point out however that deleting these lines also means that the result will not contain any prefactor, not even the one coming from the Feynman parametrisation.

6.10 How can I use complex masses?

In the python script generating the expressions for the integral, define mass symbols in the same way as for real masses, e.g:

```
Mandelstam_symbols=['s']
mass_symbols=['msq']
```

Then, in *loop_package* define:

```
real parameters = Mandelstam_symbols,
complex parameters = mass_symbols,
```

In the integration script (using the python interface), the numerical values for the complex parameters are given after the ones for the real parameters:

```
str_integral_without_prefactor, str_prefactor, str_integral_with_prefactor =␣
↪integral(real_parameters=[4.],complex_parameters=[1.-0.0038j])
```

Note that in python the letter `j` is used rather than `i` for the imaginary part.

In the C++ interface, you can set (for the example *triangle2L*):

```
const std::vector<triangle2L::real_t> real_parameters = { 4. };
const std::vector<triangle2L::complex_t> complex_parameters = { {1.,0.0038} };
```

6.11 When should I use the "split" option?

The modules *loop_package* and *make_package* have the option to split the integration domain (`split=True`). This option can be useful for integrals which do not have a Euclidean region. If certain kinematic conditions are fulfilled, for example if the integral contains massive on-shell lines, it can happen that singularities at $x_i = 1$ remain in the $\mathcal{F}$ polynomial after the decomposition. The split option remaps these singularities to the origin of parameter space. If your integral is of this type, and with the standard approach the numerical integration does not seem to converge, try the `split` option. It produces a lot more sectors, so it should not be used without need. We also would like to mention that very often a change of basis to increase the (negative) power of the $\mathcal{F}$ polynomial can be beneficial if integrals of this type occur in the calculation.

6.12 How can I obtain results from pySecDec in a format convenient for GiNaC/ Sympy/ Mathematica/ Maple?

If you are using the python interface, you can use the functions *series_to_ginac*, *series_to_sympy*, *series_to_mathematica*, *series_to_maple* to convert the output of the integral library.

Example:

```
#!/usr/bin/env python3
from pySecDec.integral_interface import IntegralLibrary
from pySecDec.integral_interface import series_to_ginac, series_to_sympy, series_to_
↪mathematica, series_to_maple
```

(continues on next page)

(continued from previous page)

```
if __name__ == "__main__":

    # load c++ library
    easy = IntegralLibrary('easy/easy_pylink.so')

    # integrate
    _, _, result = easy()

    # print result
    print(series_to_ginac(result))
    print(series_to_sympy(result))
    print(series_to_mathematica(result))
    print(series_to_maple(result))
```

Outputs:

```
('(1+0*I)/eps + (0.30685281944005254 9+0*I) + Order(eps)', '(5.41537065611170534e-17+0*I)/
↪eps + (1.3864926114078559e-15+0*I) + Order(eps)')
('(1+0*I)/eps + (0.30685281944005254 9+0*I) + O(eps)', '(5.41537065611170534e-17+0*I)/eps␣
↪+ (1.3864926114078559e-15+0*I) + O(eps)')
('(1+0*I)/eps + (0.30685281944005254 9+0*I) + O[eps]', '(5.41537065611170534*10^-17+0*I)/
↪eps + (1.3864926114078559*10^-15+0*I) + O[eps]')
('(1+0*I)/eps + (0.30685281944005254 9+0*I) + O(eps)', '(5.41537065611170534e-17+0*I)/eps␣
↪+ (1.3864926114078559e-15+0*I) + O(eps)')
```

6.13 Expansion by regions: what does the parameter z mean?

When expansion by regions via the "rescaling with z-method" is used, the parameter `z` acts as expansion parameter in the Taylor expansion of the integrand. After the code generation step, in the numerical integration, `z=1` needs to be used and the kinematic invariants have to be set to the same values as would be used with the t-method, i.e. the kinematic values desired by the user.

6.14 Expansion by regions: why does the t-method not converge?

With the t-method, configurations can occur for particular kinematic points which, after sector decomposition, lead to a pole at the upper integration boundary, where the contour deformation vanishes and therefore cannot regulate this pole. In such a case the z-method should be used, because it does not transform the Feynman parameters in a way which can induce such a configuration.

CHAPTER

SEVEN

REFERENCES

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[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

[BIS16] W. Bruns, B. Ichim, C. Söger, *The power of pyramid decomposition in Normaliz*, 2016, J.Symb.Comp.74, 513–536,
doi:10.1016/j.jsc.2015.09.003,
arXiv:1206.1916

[Bor14] S. Borowka, *Evaluation of multi-loop multi-scale integrals and phenomenological two-loop applications*, 2014, PhD Thesis - Technische Universität München
mediaTUM:1220360,
arXiv:1410.7939

[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

[GSL] M. Galassi, J. Davies, J. Theiler, B. Gough, G. Jungman, P. Alken, M. Booth, F. Rossi, *GNU Scientific Library Reference Manual - Third Edition*, 2009, Network Theory Ltd.,
ISBN: 0-9546120-7-8 (ISBN-13: 978-0-9546120-7-8),
available at http://www.gnu.org/software/gsl/

[Hah05] T. Hahn, *CUBA: A Library for multidimensional numerical integration*, 2005, Comput.Phys.Comm.168, 78-95,
doi:10.1016/j.cpc.2005.01.010,
arXiv:hep-ph/0404043

[Hah16] T. Hahn, *Concurrent Cuba*, 2016, Comput.Phys.Comm.207, 341-349,
doi:10.1016/j.cpc.2016.05.012,

arXiv:1408.6373

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

[KU10] T. Kaneko and T. Ueda, *A Geometric method of sector decomposition*, 2010, Comput.Phys.Comm.181, doi:10.1016/j.cpc.2010.04.001, arXiv:0908.2897

[KUV13] J. Kuipers, T. Ueda, J. A. M. Vermaseren, *Code Optimization in FORM*, 2015, Comput.Phys.Comm.189, 1-19, doi:10.1016/j.cpc.2014.08.008, arXiv:1310.7007

[LWY+15] Z. Li, J. Wang, Q.-S. Yan, X. Zhao, *Efficient Numerical Evaluation of Feynman Integrals*, 2016, Chin.Phys.C40 No. 3, 033103, doi:10.1088/1674-1137/40/3/033103, arXiv:1508.02512

[MP+14] B. D. McKay and A. Piperno, *Practical graph isomorphism, II*, 2014, Journal of Symbolic Computation, 60, 94-112, doi:10.1016/j.jsc.2013.09.003 arXiv:1301.1493

[Pak11] A. Pak, *The toolbox of modern multi-loop calculations: novel analytic and semi-analytic techniques*, 2012, J. Phys.: Conf. Ser. 368 012049, doi:10.1088/1742-6596/368/1/012049, arXiv:1111.0868

[PS11] A. Pak, A. Smirnov, *Geometric approach to asymptotic expansion of Feynman integrals*, 2011, Eur.Phys.J.C 71, 1626, doi:10.1140/epjc/s10052-011-1626-1, arXiv:1011.4863

[PSD17] S. Borowka, G. Heinrich, S. Jahn, S. P. Jones, M. Kerner, J. Schlenk, T. Zirke, *pySecDec: A toolbox for the numerical evaluation of multi-scale integrals*, Comput.Phys.Comm. 222 (2018), doi:10.1016/j.cpc.2017.09.015, arXiv:1703.09692

[PSD18] S. Borowka, G. Heinrich, S. Jahn, S. P. Jones, M. Kerner, J. Schlenk, *A GPU compatible quasi-Monte Carlo integrator interfaced to pySecDec*, Comput.Phys.Commun. 240 (2019), doi:10.1016/j.cpc.2019.02.015, arXiv:1811.11720

[PSD21] G. Heinrich, S. Jahn, S. P. Jones, M. Kerner, F. Langer, V Magerya, A Poldaru, J. Schlenk, E. Villa *Expansion by regions with pySecDec*, (to appear)

[RUV17] B. Ruijl, T. Ueda, J. Vermaseren, *FORM version 4.2*, arXiv:1707.06453

[Ver00] J. A. M. Vermaseren, *New features of FORM*, arXiv:math-ph/0010025

[Mis18] G. Mishima, *High-Energy Expansion of Two-Loop Massive Four-Point Diagrams* doi:10.1007/JHEP02(2019)080,

arXiv:1812.04373

PYTHON MODULE INDEX

INDEX

G

H

I

L

M

N

O

P

Q

R

S

T

V