Package **susy_cross_section** Manual

Sho Iwamoto / Misho

Dipartimento di Fisica e Astronomia, Università degli Studi di Padova
Via Marzolo 8, I-35131 Padova, Italia

sho.iwamoto@pd.infn.it

Abstract

susy_cross_section is a Python package to handle cross-section grid tables regardless of their format. This package may parse any table-like grid files as a pandas.DataFrame object, and with built-in interpolators users can get interpolated values of cross sections together with uncertainties. Several table data is also pre-installed in this package.

1. Introduction

1.1. Background

Collider experiments have long been the central tools to look for new physics beyond the Standard Model (SM). The number of new physics events in colliders are given by the product of production cross section and an integrated luminosity, where the former is theoretically

calculated based on a hypothesized theory and the latter is measured by the experiment collaboration with, e.g., at the ATLAS or CMS experiments at the LHC, 2%-level precision [1,2]. Therefore, cross sections of new physics processes are the most important values in any new-physics theory and should be calculated with similar precision.

Cross-section calculation with such precision is not a simple task because the leading order (LO) calculation, which usually includes only the tree-level contributions, will not give such precision and we have to include loop-level calculations. Especially, when colored particles are involved in the process, the large QCD couplings worsen convergence of the perturbation series and even the next-to-leading-order (NLO) calculation may give uncertainties more than 10%, and we have to include the next-to-NLO (NNLO) diagrams and/or soft-gluon resummation.

For SUSY processes, several tools are published for precise cross-section calculations. Prospino [3] is one of the pioneer works. It is upgraded to Prospino 2 [4], with which we can calculate NLO cross sections of most SUSY processes within a few minutes. For soft-gluon resummation, Resummino [5] is available, which allows us to calculate the resummation at the accuracy level of next-to-leading-log (NLL) or the next-to-NLL (NNLL).

NNLL-fast [6] is another type of tools for SUSY cross sections. It provides grid data of cross sections calculated with accuracy level of approximated-NNLO plus NNLL together with an interpolator, with which users can obtain cross sections for various parameter set of simplified scenario less than a few second. While available process are limited compared to Prospino 2 or Resummino and the results suffer from uncertainties due to the interpolation, the fast calculation provided by NNLL-fast has a great advantage for new-physics studies, which usually consider a huge number of parameter sets.

Grid tables for SUSY cross sections are provided by other collaborations as well[*1]. The most nominal set is the one provided by LHC SUSY Cross Section Working Group [8], which is obtained by compiling the results from the above calculators. They provide cross-section grid tables for various simplified models and collision energies. However, the grid tables provided by various collaborations are (of course) in various formats. This package `susy_cross_section` aims to handle those grid data, including ones appearing in future, in an unified manner.

1.2. This Package

`susy_cross_section` is a Python package to handle cross-section grid tables regardless of their format. With this package, one can import any table-like grid files as a pandas DataFrame, once an annotation file (`info` file) is provided in JSON format [9]. Several interpolators are also provided, with which one can interpolate the grid tables to obtain the central values together with (possibly asymmetric) uncertainties. A quick start guide is provided in Section 2.

For simple use-cases, a command-line script `susy-xs` is provided. You can get interpolated cross sections for simplified scenarios with the sub-command `susy-xs get` on your terminal, based on simple log-log interpolators. More information on the script is available in Section 3.

[*1] DeepXS [7] is another tool for precise SUSY cross section, which utilizes deep learning technique for cross-section estimation.

You can include this package in your Python code for more customization. For example, you may use the cross section values in your code, or interpolate the grid tables with other interpolators, including your own ones. Section 4 of this document is devoted to such use-cases.

The appendices contain additional information of this package. Appendix A is the full API reference of this package, and in Appendix B two methods to estimate the errors due to interpolation are provided together with the validation results.

2. Quick Start

2.1. Install and uninstall

This package requires Python with version 2.7, 3.5 or above with `pip` package manager. You may check the versions by

```
$ python -V
Python 3.6.7
$ pip -V
pip 19.0.3
```

With the set-up, you can simply install this package via PyPI:

```
$ pip install susy-cross-section          # for install
$ pip install --upgrade susy-cross-section  # or for upgrade

Collecting susy-cross-section
...
Successfully installed susy-cross-section-(version)
```

You can also instantly uninstall this package by

```
$ pip uninstall susy-cross-section
```

2.2. Run command-line script

This package provides a command line script `susy-xs`. With using a keyword `13TeV.n2x1+.wino`, you can get the production cross sections of neutralino–chargino pair at 13 TeV LHC, $\sigma_{13\text{TeV}}(pp \to \tilde{\chi}_2^0 \tilde{\chi}_1^+)$:

```
$ susy-xs get 13TeV.n2x1+.wino 500
(32.9 +2.7 -2.7) fb

$ susy-xs get 13TeV.n2x1+.wino 513.3
(29.4 +2.5 -2.5) fb

$ susy-xs get 13TeV.n2x1+.wino
Usage: get [OPTIONS] 13TeV.n2x1+.wino M_WINO
    Parameters: M_WINO    [unit: GeV]
    Table-specific options: --name=xsec   [unit: fb]  (default)
```

Here, as you may guess, the neutralino $\tilde{\chi}_2^0$ and the chargino $\tilde{\chi}_1^+$ are assumed to be wino-like and degenerate in mass: they are 500 GeV in the first command, while 513.3 GeV in the second command. As shown in the third command, calling `get` sub-command without the mass parameter shows a short help, where you can see `13TeV.n2x1+.wino` accepts one argument `M_WINO` in the unit of GeV and by default returns `xsec` in the unit of fb.

For more information, you may run `show` sub-command:

```
$ susy-xs show 13TeV.n2x1+.wino

----------------------------------------------------------------------
TABLE "xsec" (unit: fb)
----------------------------------------------------------------------
            value        unc+        unc-
m_wino
100     13895.100000  485.572000  485.572000
125      6252.210000  222.508000  222.508000
150      3273.840000  127.175000  127.175000
...             ...         ...         ...
475        41.023300    3.288370    3.288370
500        32.913500    2.734430    2.734430
525        26.602800    2.299570    2.299570
...             ...         ...         ...
1950        0.005096    0.001769    0.001769
1975        0.004448    0.001679    0.001679
2000        0.003892    0.001551    0.001551


[77 rows x 3 columns]

collider: pp-collider, ECM=13TeV
calculation order: NLO+NLL
PDF: Envelope by LHC SUSY Cross Section Working Group
included processes:
  p p > wino0 wino+


----------------------------------------------------------------------
title: NLO-NLL wino-like chargino-neutralino (N2C1) cross sections
authors: LHC SUSY Cross Section Working Group
calculator: resummino
source: https://twiki.cern.ch/twiki/bin/view/LHCPhysics/SUSYCrossSections13TeVn2x1wino
version: 2017-06-15
----------------------------------------------------------------------
```

Here, you see the `xsec` grid table followed by physical parameters and documental information. You may also notice that the above-shown result at 500 GeV is simply taken from the grid data, while that an interpolation is performed to get the cross section of 513.3 GeV wino.

You can list-up all the available tables, or search for a table you want, by `list` sub-command:

```
$ susy-xs list  # to list up all the (tagged) tables.
13TeV.n2x1-.wino       lhc_susy_xs_wg/13TeVn2x1wino_envelope_m.csv
13TeV.n2x1+.wino       lhc_susy_xs_wg/13TeVn2x1wino_envelope_p.csv
13TeV.n2x1+-.wino      lhc_susy_xs_wg/13TeVn2x1wino_envelope_pm.csv
13TeV.slepslep.ll      lhc_susy_xs_wg/13TeVslepslep_ll.csv
13TeV.slepslep.maxmix  lhc_susy_xs_wg/13TeVslepslep_maxmix.csv
```

```
13TeV.slepslep.rr      lhc_susy_xs_wg/13TeVslepslep_rr.csv
...

$ susy-xs list 7TeV  # to show tables including '7TeV' in its key or paths.
7TeV.gg.decoup  nllfast/7TeV/gdcpl_nllnlo_mstw2008.grid
7TeV.gg.high    nllfast/7TeV/gg_nllnlo_hm_mstw2008.grid
7TeV.gg         nllfast/7TeV/gg_nllnlo_mstw2008.grid
...
7TeV.ss10       nllfast/7TeV/ss_nllnlo_mstw2008.grid
7TeV.st         nllfast/7TeV/st_nllnlo_mstw2008.grid


$ susy-xs list 8t decoup
8TeV.gg.decoup    nllfast/8TeV/gdcpl_nllnlo_mstw2008.grid
8TeV.sb10.decoup  nllfast/8TeV/sdcpl_nllnlo_mstw2008.grid
```

Then you will run, for example,

```
$ susy-xs get 8TeV.gg.decoup
Usage: get [OPTIONS] 8TeV.gg.decoup MGL
    Parameters: MGL   [unit: GeV]
    Table-specific options: --name=xsec_lo   [unit: pb]
                            --name=xsec_nlo  [unit: pb]
                            --name=xsec      [unit: pb]  (default)

$ susy-xs get 8TeV.gg.decoup --name=xsec_lo 1210
(0.00207 +0.00100 -0.00065) pb
$ susy-xs get 8TeV.gg.decoup --name=xsec 1210
(0.00325 +0.00055 -0.00051) pb
```

More information is available with `--help` options:

```
$ susy-xs --help
$ susy-xs get --help
$ susy-xs show --help
$ susy-xs list --help
```

2.3. Use as a package

The above results are obtained also in your Python code. For example,

```python
from susy_cross_section import utility
from susy_cross_section.table import File, Table
from susy_cross_section.interp.interpolator import Scipy1dInterpolator

grid_path, info_path = utility.get_paths("13TeV.n2x1+.wino")
file = File(grid_path, info_path)

document = file.info.document
print(document)

xsec_table = file["xsec"]
```

6

```
xsec_attr = xsec_table.attributes
print(xsec_attr.formatted_str())
```

will show the documents and attributes, and you may interpolate the table by

```
interpolator = Scipy1dInterpolator(axes="loglog", kind="spline")
xs = interpolator.interpolate(xsec_table)
print(xs(500), xs.fp(500), xs.fm(500), xs.unc_p_at(500), xs.unc_m_at(500))
print(xs.tuple_at(513.3))
```

The output will be something like this, which reproduces the above-obtained results:

```
32.9135     35.6479     30.1791     2.7344       -2.7344
(array(29.3516), 2.4916, -2.4916)
```

Note that the interpolator is `Scipy1dInterpolator` with `spline` option in log-log axes. You may use another interpolator, such as linear interpolator in log-log axes, by

```
pchip = Scipy1dInterpolator(axes="loglog", kind="linear").interpolate(xsec_table)
print(pchip.tuple_at(500))
print(pchip.tuple_at(513.3))
```

The output will be:

```
(array(32.9135), 2.7344, -2.7344)
(array(29.3641), 2.4932, -2.4932)
```

The results for 500 GeV is the same because it is on the grid and without interpolation, but the values for 513.3 GeV are slightly different from the previous ones.

More information is available in API references.

3. Usage: as a command-line script

The package provides one script for terminal, `susy-xs`, which accepts the following flags and sub-commands.

- `susy-xs --help` gives a short help and a list of sub-commands,

- `susy-xs --version` returns the package version,

- `susy-xs list` displays a list of available table-grid data files,

- `susy-xs show` shows the information of a specified data file,

- `susy-xs get` obtains a cross section value from a table, with interpolation if necessary.

Details of these sub-commands are explained below, or available from the terminal with `--help` flag as, for example, `susy-xs get --help`.

7

3.1. list

```
$ susy-xs list (options) (substr substr ...)
```

This sub-command displays a list of available cross-section tables. If `substr` is specified, only the tables which includes it in the table name or file paths are displayed.

By default, this command lists only the files with pre-defined table keys. In addition to these commonly-used table grids, this package contains much more cross-section data. One can find these additional files with an option `--all`.

With `--full` option, full paths to the files are displayed, which is useful for additional operations, for example,

```
$ susy-xs list --all --full gg 7TeV CTEQ
/Users/misho/ (...) /data/nllfast/7TeV/gg_nllnlo_cteq6.grid
/Users/misho/ (...) /data/nllfast/7TeV/gg_nllnlo_hm_cteq6.grid

$ susy-xs show /Users/misho/ (...) /data/nllfast/7TeV/gg_nllnlo_hm_cteq6.grid
--------------------------------------------------------------------
TABLE "xsec_lo" (unit: pb)
--------------------------------------------------------------------
                value           unc+          unc-
ms    mgl
200   200    3.400000e+02   1.411437e+02   9.385184e+01
...
```

3.2. show

```
$ susy-xs show (options) table
```

This sub-command shows data and information of the table specified by `table`. `table` can be one of pre-defined table keys, which can be displayed by *list sub-command*, or a path to grid-data file. The displayed information includes grid-tables in the file, physical attributes associated to each of the tables, and the documenting information associated to the file.

A grid-data file is read with an associated "info" file, whose name is by default resolved by replacing the suffix of the data file to `.info`. One can override this default behavior with the `--info` option.

3.3. get

```
$ susy-xs get (options) table (args ...)
```

This sub-command gets a cross-section value from the table specified by `table` and the option `--name`, where `args` are used as the physical parameters. Without `args`, this sub-command displays the meanings of `args` and `--name` option, such as

```
$ susy-xs get 8TeV.gg
Usage: get [OPTIONS] 8TeV.gg MS MGL

Parameters: MS    [unit: GeV]
            MGL   [unit: GeV]

Table-specific options: --name=xsec_lo    [unit: pb]
                        --name=xsec_nlo   [unit: pb]
                        --name=xsec       [unit: pb]  (default)
```

In this case, users are asked to specify the squark mass (which is assumed to be degenerate in this grid) as the first `args` and gluino mass as the second `args`, both in GeV. It is also shown here that users can get LO and NLO cross sections by using `-name` option or otherwise the default `xsec` grid is used. So, for example, the cross section $\sigma_{8\text{TeV}}(pp \to \tilde{g}\tilde{g})$ with 1 TeV gluino and 1.2 TeV squark can be obtained by

```
$ susy-xs get 8TeV.gg 1200 1000
(0.0126 +0.0023 -0.0023) pb
```

Here, the default `xsec` grid in the table file `8TeV.gg` is used. One can check with *show sub-command* that this grid is calculated by NLL-fast collaboration at the NLO+NLL order with using MSTW2008nlo68cl as the parton distribution functions (PDFs), and thus this 12.6 fb is the NLO+NLL cross section.

The value is calculated by an interpolation if necessary. This sub-command uses (possibly multi-dimensional) cubic spline interpolation in logarithmic scale of all the parameters and value because the interpolator tends to give the best results in validations. To use other interpolating methods, users have to use this package by importing it to their Python codes as explained in Section 4. For details, confer the API document of `susy_cross_section.interp`.

`table` can be one of pre-defined table keys, which can be displayed by *list sub-command*, or a path to grid-data file. A grid-data file is read with an associated "info" file, whose name is by default resolved by replacing the suffix of the data file to `.info`. One can override this default behavior with the `--info` option.

Additionally, several options are provided to control the output format, which are found in the `--help`.

> **Caution:** Theoretically, one can get cross sections for various model point by repeating this sub-command. However, it is not recommended since this sub-command construct an interpolating function every time. For such use-cases, users should use this package as a package, i.e., import this package in their Python codes, as explained in Section 4.

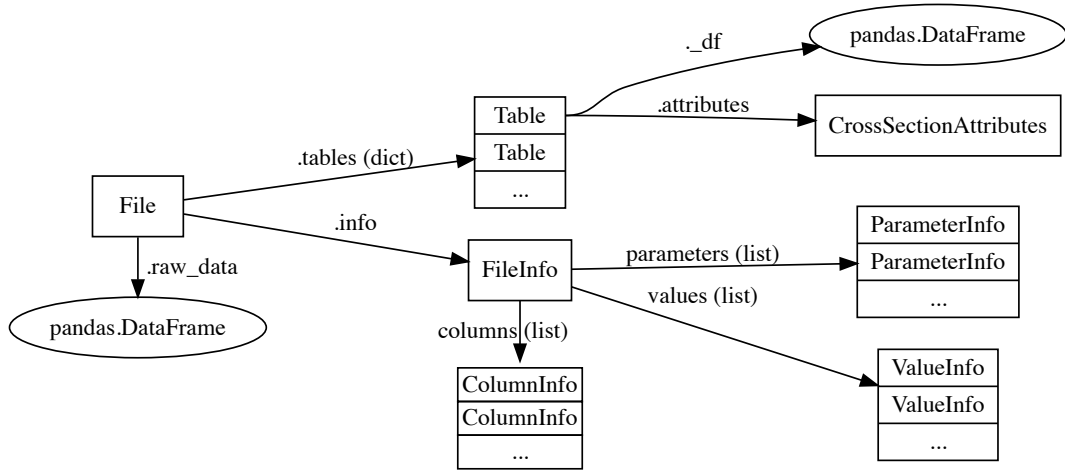Figure 1: Conceptual structure of data and classes.

4. Usage: as a Python package

4.1. Grid-data file and Info file

The fundamental objects of this package are `File` and `Table` classes, representing the files and the cross-section grid tables, respectively. A `File` instance carries two files as paths: `File.table_path` for grid-data file and `File.info_path`. A grid-data file contains a table representing one or more cross sections. The content of a grid-data file is read and parsed by `pandas.read_csv`, which can parse most of table-format files[*1] with a proper `reader_options` specified in the "info" file. The resulting `pandas.DataFrame` object is stored as-is in `File.raw_table` for further interpretation.

A "info" file corresponds `FileInfo` instance and is provided in JSON format [9]. It has data for `ColumnInfo`, `ParameterInfo`, and `ValueInfo` objects in addition to `reader_options`. Those three types of information is used to interpret the `File.raw_table` data. Detailed specification of "info" files are described below.

One grid table has multiple columns, where the name and unit of each column is specified by `ColumnInfo`. Some columns are "parameters" for cross sections, such as the mass of relevant particles, which are specified by `ParameterInfo`. Other columns are for "values" and `ValueInfo` is used to define the values. `ValueInfo` uses one column as a central value, and one or more columns as uncertainties, which can be relative or absolute and symmetric or asymmetric. Multiple columns for an uncertainty are combined in quadrature, i.e., $\sigma_1 \oplus \sigma_2 := \sqrt{\sigma_1^2 + \sigma_2^2}$.

[*1] Parsable formats include comma-separated values (CSV), tab-separated values (TSV), and space-separated values (SSV); in addition, fixed-width formatted tables are usually parsable.

For each `ValueInfo`, the interpreter constructs one `DataFrame` object. It is parameterized by `Index` or `MultiIndex` and three columns, `value`, `unc+`, and `unc-`, respectively containing the cross-section central value, positive combined absolute uncertainty, and (the absolute values of) negative combined absolute uncertainty. The `DataFrame` is wrapped by `Table` class and stored in `File.tables` (*dict*) with keys being the `name` of the value columns.

This is an example of data handling:

```python
from susy_cross_section import utility
from susy_cross_section.table import File, Table

grid_path, info_path = utility.get_paths("13TeV.n2x1+.wino")
file = File(grid_path, info_path)

xsec_table = file.tables["xsec"]
```

Here an utility function `get_paths` is used to look-up paths for the key `13TeV.n2x1+.wino` and from the passes a `File` instance is constructed. Then a table with the column name `xsec` is read from the `tables` dictionary.

4.2. Interpolation

The table interpolation is handled by `susy_cross_section.interp` subpackage. This package first performs axes transformation using `axes_wrapper` module, and then use one of the interpolators defined in `interpolator` module. Detail information is available in the API document of each module.

The cross-section data with one mass parameter are usually fit well by a negative power of the mass, i.e., $\sigma(m) \propto m^{-n}$. For such cases, interpolating the function by piece-wise lines in log-log axes would work well, which is implemented as

```python
from susy_cross_section.interp.interpolator import Scipy1dInterpolator

xs = Scipy1dInterpolator(axes="loglog", kind="linear").interpolate(xsec_table)
print(xs(500), xs.fp(500), xs.fm(500), xs.unc_p_at(500), xs.unc_m_at(500))
```

One can implement more complicated interpolators by extending `AbstractInterpolator`.

4.3. A proposal for INFO file format

An info file is a JSON file and its data is one dict object. The dict has six keys: `document`, `attributes` (optional), `columns`, `reader_options` (optional), `parameters`, and `values`.

document as *dict(str, str)*:

> This dictionary may contain any values and no specification is given, but the content should be used only for documental purposes; i.e., programs should not change their behavior by the content of `document`. Data for such purposes should be stored not in `document` but in `attributes`.

Possible keys are: `title`, `authors`, `calculator`, `source`, and `version`.

attributes as *dict(str, str)*:

This dictionary contains *the default values* for `CrossSectionAttributes`, which is attached to each values. These default values are overridden by the `attributes` defined in respective values.

`CrossSectionAttributes` stores, contrary to `document`, non-documental information, based on which programs may change their behavior. Therefore the content must be neat and in machine-friendly formats. The proposed keys are: `processes`, `collider`, `ecm`, `order`, and `pdf_name`. For details, see the API document of `CrossSectionAttributes`.

columns as a list of *dict(str, str)*:

This is a list of dictionaries used to construct `ColumnInfo`; the n-th element defines n-th column in the grid-data file. The length of this list thus matches the number of the columns. Each dictionary must have two keys: `name` and `unit`, respectively specify the name and unit of the column. The names must be unique in one file. For dimension-less column, `unit` is an empty string.

reader_options as *dict(str, Any)*:

This dictionary is directly passed to `read_csv()` and used as the keyword arguments.

parameters as a list of *dict(str, Any)*:

This list defines the parameters for indexing. Each element is a dictionary, which has two keys `column` and `granularity` and constructs a `ParameterInfo` object. The value for `column` is one of the `name` of `columns`. The value for `granularity` is a number used to quantize the parameter grid; for details see the API document of `ParameterInfo`.

values as a list of dictionary:

This list defines the cross-section values. Each element is a dictionary and constructs a `ValueInfo` object. The dictionary has possibly the keys `column`, `unc`, `unc+`, `unc-`, and `attributes`. Among these keys, `column` is mandatory and corresponding value must be one of the `name` of `columns`, where the column is used as the central value of cross-section. The value for `attributes` is a dictionary *dict(str, Any)*. It overrides the file-wide default values (explained above) to construct a `CrossSectionAttributes`.

The other three keys are used to specify uncertainties. `unc` specifies symmetric uncertainty, while a pair of `unc+` and `unc-` specifies asymmetric uncertainty; `unc` will not be present together with `unc+` or `unc-`. Each value of `unc`, `unc+`, and `unc-` is *a list of dictionaries*, *list(dict(str, str))*. Each element of the list, being a dictionary with two keys `column` and `type`, describes one source of uncertainties. The value for `column` is one of the `name` of `columns`, or a list of the names. If one name is specified, the column is used as the source. If a list is specified, the column with the largest value among them is used as the source. The value for `type` specifies the type of uncertainty; possible options and further details are found in the API document of `ValueInfo`.

12

4.4. How to use own tables

Users may use this package to handle their own cross-section grid tables, once they provide an INFO file. The procedure is summarized as follows.

1. Find proper `reader_options` to read the table.

   This package uses `pandas.read_csv()` to read the grid table, for which proper options should be specified. The following script may be useful to find the proper option for your table. Possible keys for `reader_options` are found in the API document of `pandas.read_csv()`.

```python
import pandas

reader_options = {
    "sep": ";",
    "skiprows": 1
}
grid_path = "mydata/table_grid.txt"
data_frame = pandas.read_csv(grid_path, **reader_options)
print(data_frame)
```

2. Write the INFO file. One should be careful especially of "type" of uncertainties and "unit" of columns.

3. Verify whether the file is correctly read. *show sub-command* is useful for this purpose; for example,

```
$ susy-xs show mydata/table_grid.txt mydata/table_grid.info
```

   After verifying with show sub-command, users can use *get sub-command*, or read the data in their code as:

```
my_grid = File("mydata/table_grid.txt", "mydata/table_grid.info")
```

5. Conclusion

This paper introduces a Python package "`susy_cross_section`" to handle cross-section grid tables provided by various groups in various format. Users can install it via `PyPI` to load the grid tables, to get cross-section values by interpolations, and to validate the interpolating function, in an unified manner in their Python codes. A terminal command "`susy-xs`" is provided, with which users can glance the grid data and cross-section values. Several grid tables are pre-installed, which are originally calculated and distributed by NNLL-fast collaboration and LHC SUSY Cross Section Working Group.

This package is managed on GitHub: `misho104/susy_cross_section`. Bug reports, comments, questions, and contribution are welcome on the website.

Acknowledgments

A. API Reference: `susy_cross_section`

Module to handle CSV-like data of SUSY cross section.

A.1. `susy_cross_section.base` subpackage

Table of values with asymmetric uncertainties.

`BaseFile` carries grid tables as `BaseTable` and annotations as `FileInfo`. The `FileInfo` class contains the other three classes as sub-information.

base.table.BaseFile	contains `FileInfo` and multiple `BaseTable`
base.table.BaseTable	represents the grid data for cross section.
base.info.FileInfo	has file-wide properties and `ColumnInfo`, `ParameterInfo`, and `ValueInfo`
base.info.ColumnInfo	has properties of each column
base.info.ParameterInfo	annotates a column as a parameter
base.info.ValueInfo	defines a value from columns

A.1.1. `susy_cross_section.base.info` module

Classes to describe annotations of general-purpose tables.

This module provides annotation classes for CSV-like table data. The data is a two-dimensional table and represents functions over a parameter space. Some columns represent parameters and others do values. Each row represents a single data point and corresponding value.

Two structural annotations and two semantic annotations are defined. `FileInfo` and `ColumnInfo` are structural, which respectively annotate the whole file and each columns. For semantics, `ParameterInfo` collects the information of parameters, each of which is a column, and `ValueInfo` is for a value. A value may be given by multiple columns if, for example, the value has uncertainties or the value is given by the average of two columns.

class `ColumnInfo`(*index, name, unit=''*)

>   Bases: `object`
>
>   Stores information of a column.
>
>   Instead of the *int* identifier `index`, we use `name` as the principal identifier for readability. We also annotate a column by `unit`, which is *str* that is passed to `Unit()`.
>
>   Variables
>
>   - **index** (*int*) – The zero-based index of column.
>
>   The columns of a file should have valid `index`, i.e., no overlap, no gap, and starting from zero.
>
>   - **name** (*str*) – The human-readable and machine-readable name of the column.
>
>   As it is used as the identifier, it should be unique in one file.
>
>   - **unit** (*str*) – The unit of column, or empty string if the column has no unit.
>
>   The default value is an empty str `''`, which means the column has no unit. Internally this is passed to `Unit()`.

Note: As for now, `unit` is restricted as a str object, but in future a float should be allowed to describe "x1000" etc.

classmethod from_json(*json_obj*)

    Initialize an instance from valid json data.

    Parameters **json_obj** (*Any*) – a valid json object.

    Returns ColumnInfo – Constructed instance.

    Raises `ValueError` – If json_obj has invalid data.

to_json()

    Serialize the object to a json data.

    Returns *dict(str, str or int)* – The json data describing the object.

validate()

    Validate the content.

    Raises

- `TypeError` – If any attributes are invalid type of instance.
- `ValueError` – If any attributes have invalid content.

class ParameterInfo(*column='', granularity=None*)

    Bases: `object`

Stores information of a parameter.

A parameter set defines a data point for the functions described by the file. A parameter set has one or more parameters, each of which corresponds to a column of the file. The `column` attribute has `ColumnInfo.name` of the column.

Since the parameter value is read from an ASCII file, *float* values might have round-off errors, which might cause grid misalignments in grid- based interpolations. To have the same *float* expression on the numbers that should be on the same grid, `granularity` should be provided.

Variables

- **column** (*str*) – Name of the column that stores this parameter.
- **granularity** (*int or float, optional*) – Assumed presicion of the parameter.

  This is used to round the parameter so that a data point should be exactly on the grid. Internally, a parameter is rounded to:

  ```
  round(value / granularity) * granularity
  ```

  For example, for a grid `[10, 20, 30, 50, 70]`, it should be set to 10 (or 5, 1, 0.1, etc.), while for `[33.3, 50, 90]`, it should be 0.01.

classmethod from_json(*json_obj*)

    Initialize an instance from valid json data.

    Parameters **json_obj** (*Any*) – a valid json object.

    Returns ParameterInfo – Constructed instance.

    Raises `ValueError` – If json_obj has invalid data.

to_json()
> Serialize the object to a json data.
>
> Returns *dict(str, str or float)* – The json data describing the object.

validate()
> Validate the content.
>
> Raises
>
> > - `TypeError` – If any attributes are invalid type of instance.
> > - `ValueError` – If any attributes have invalid content.

class ValueInfo(*column='', attributes=None, unc_ p=None, unc_ m=None*)
> Bases: `object`

Stores information of value accompanied by uncertainties.

A value is generally composed from several columns. In current implementation, the central value must be given by one column, whose name is specified by `column`. The positive- and negative-direction uncertainties are specified by `unc_p` and `unc_m`, respectively, which are *dict(str, str)*.

Variables

- **column** (*str or List[str]*) – Names of the column that stores this value.

  The string, or each element of the list, must match one of the `ColumnInfo.name` in the file. If multiple columns are specified, the largest value among the columns (compared in each row) is used.

- **attributes** (*dict (str, Any)*) – Physical information annotated to this value.

- **unc_p** (*dict (str, str)*) – The sources of "plus" uncertainties.

  Multiple uncertainty sources can be specified. Each key corresponds `ColumnInfo.name` of the source column, and each value denotes the "type" of the source. Currently, two types are implementend:

  - `"relative"` for relative uncertainty, where the unit of the column must be dimensionless.

  - `"absolute"` for absolute uncertainty, where the unit of the column must be the same as that of the value column up to a factor.

  - `"absolute,signed"` or `"relative,signed"` for absolute/relative uncertainty but using the columns with correct sign.

- **unc_m** (*dict(str, str)*) – The sources of "minus" uncertainties.

  Details are the same as `unc_p`.

validate()
> Validate the content.

classmethod from_json(*json_ obj*)
> Initialize an instance from valid json data.
>
> Parameters **json_obj** (*typing.Any*) – a valid json object.
>
> Returns ValueInfo – Constructed instance.
>
> Raises `ValueError` – If json_obj has invalid data.

to_json()
> Serialize the object to a json data.
>
> Returns *dict(str, str or float)* – The json data describing the object.

class FileInfo(*document=None,* *columns=None,* *parameters=None,* *values=None,* *reader_options=None*)
> Bases: object
>
> Stores file-wide annotations.
>
> A table structure is given by `columns`, while in semantics a table consists of `parameters` and `values`. The information about them is stored as lists of `ColumnInfo`, `ParameterInfo`, and `ValueInfo` objects. In addition, `reader_options` can be specified, which is directly passed to `pandas.read_csv()`.
>
> The attribute `document` is provided just for documentation. The information is guaranteed not to modify any functionality of codes or packages, and thus can be anything.
>
> Developers must not use `document` information except for displaying them. If one needs to interpret some information, one should extend this class to provide other data-storage for such information.
>
> Variables
>
> - **document** (*dict(Any, Any)*) – Any information for documentation without physical meanings.
>
> - **columns** (*list of ColumnInfo*) – The list of columns.
>
> - **parameters** (*list of ParameterInfo*) – The list of parameters to define a data point.
>
> - **values** (*list of ValueInfo*) – The list of values described in the file.
>
> - **reader_options** (*dict(str, Any)*) – Options to read the CSV
>
>   The values are directly passed to `pandas.read_csv()` as keyword arguments, so all the options of `pandas.read_csv()` are available.

validate()
> Validate the content.

classmethod load(*source*)
> Load and construct FileInfo from a json file.
>
> Parameters **source** (*pathlib.Path or str*) – Path to the json file.
>
> Returns FileInfo – Constructed instance.

get_column(*name*)
> Return a column with specified name.
>
> Return `ColumnInfo` of a column with name name.
>
> Parameters **name** – The name of column to get.
>
> Returns ColumnInfo – The column with name name.
>
> Raises `KeyError` – If no column is found.

formatted_str()
> Return the formatted string.
>
> Returns *str* – Dumped data.

A.1.2. `susy_cross_section.base.table` module

Tables representing values with asymmetric uncertainties.

This module provides a class to handle CSV-like table data representing values with asymmetric uncertainties. Such tables are provided in various format; for example, the uncertainty may be relative or absolute, or with multiple sources. The class `BaseFile` interprets such tables based on `FileInfo` annotations.

class `BaseTable`(*obj=None, file=None, name=None*)
>   Bases: `object`

>   Table object with annotations.

>   This is a wrapper class of `pandas.DataFrame`. Any methods except for read/write of `file` are delegated to the DataFrame object.

>   Variables

>   >   - **`file`** (*BaseFile, optional*) – File object containing this table.

>   >   - **`name`** (*str, optional*) – Name of this table.

>   >   This is provided so that `ValueInfo` can be obtained from `file`.

>   **`__getattr__`**(*name*)
>   >   Fall-back method to delegate any operations to the DataFrame.

>   **`__setitem__`**(*name, obj*)
>   >   Perform DataFrame.__setitem__.

>   **`__getitem__`**(*name*)
>   >   Perform DataFrame.__getitem__.

>   **`__str__`**()
>   >   Dump the data-frame.

class `BaseFile`(*table_path, info_path=None*)
>   Bases: `typing.Generic`

>   File with table data-sets and annotations.

>   An instance has two main attributes: `info` (*FileInfo*) as the annotation and `tables` (*dict* of *BaseTable*) as the data tables.

>   Parameters

>   >   - **`table_path`** (*str or pathlib.Path*) – Path to the csv data file.

>   >   - **`info_path`** (*str or pathlib.Path, optional*) – Path to the corresponding info file.

>   >   If unspecified, `table_path` with suffix changed to ".info" is used.

>   Variables

>   >   - **`table_path`** (*pathlib.Path*) – Path to the csv data file.

>   >   - **`info_path`** (*pathlib.Path*) – Path to the info file.

>   >   - **`raw_data`** (*pandas.DataFrame*) – the content of `table_path`.

>   >   - **`info`** (*FileInfo*) – the content of `info_path`.

- **tables** (*dict(str, BaseTable)*) – The table parsed according to the annotation.

  Each value is practically a `pandas.DataFrame` object and indexed according to the parameter specified in `info`, having exactly three value-columns: `"value"`, `"unc+"`, and `"unc-"` for the central value and positive- and negative- directed **absolute** uncertainty, respectively. The content of `"unc-"` is non-positive.

validate()
    Validate the Table data.

__getitem__(*key*)
    Return the specied table data.

    Parameters **key** (*str*) – One of The key of the data to return.

    Returns *pandas.DataFrame* – One of the data tables specified by key.

dump(*keys=None*)
    Return the dumped string of the data tables.

    Parameters **keys** (*list of str, optional*) – if specified, specified data are only dumped.

    Returns *str* – Dumped data.

A.2. `susy_cross_section.table` module

Classes for annotations to a table.

CrossSectionAttributes	represents physical property of cross section.
Table	extends `BaseTable` to handle cross-section specific attributes
File	extends `BaseFile` to carry `Table` objects.

class CrossSectionAttributes(*processes='', collider='', ecm='', order='', pdf_name=''*)
    Bases: `object`

Stores physical attributes of a cross section table.

These information is intended to be handled by program codes, so the content should be neat, clear, and ready to be standardized.

Variables

- **processes** (*list of str*) – The processes included in the cross section values. MadGraph5 syntax is recommended. Definiteness should be best respected.

- **collider** (*str*) – The type of collider, e.g., `"pp"`, `"e+e-"`.

- **ecm** (*str*) – The initial collision energy with unit.

- **order** (*str*) – The order of the cross-section calculation.

- **pdf_name** (*str*) – The name of PDF used in calculation.

  The LHAPDF's set name is recommended.

validate()
    Validate the content.

    Type is also strictly checked in order to validate info files.

    Raises

- **TypeError** – If any attributes are invalid type of instance.

- **ValueError** – If any attributes have invalid content.

formatted_str()
>   Return the formatted string.

>   Returns *str* – Dumped data.

class Table(*obj=None, file=None, name=None*)
>   Bases: susy_cross_section.base.table.BaseTable

>   Table object with annotations.

>   **__str__()**
>   >   Dump the data-frame with information.

>   **unit**
>   >   Return the unit of table values.

>   **attributes**
>   >   Return the information associated to this table.

class File(*table_path, info_path=None*)
>   Bases: susy_cross_section.base.table.BaseFile

>   Data of a cross section with parameters, read from a table file.

>   Contents are the same as superclass but each table is extended from BaseTable to Table class.

>   Variables

>   - **table_path** (*pathlib.Path*) – Path to the csv data file.

>   - **info_path** (*pathlib.Path*) – Path to the info file.

>   - **raw_data** (*pandas.DataFrame*) – the content of table_path.

>   - **info** (*FileInfo*) – the content of info_path.

>   - **tables** (*dict(str, Table)*) – The cross-section table parsed according to the annotation.

A.3. susy_cross_section.interp subpackage

A subpackage to perform interpolation.

At the subpackage-level, the following modules and class aliases are defined.

module interp.axes_wrapper	has axis preprocessors for interpolation
module interp.interpolator	has interpolator classes
interp.Scipy1dInterpolator	= interp.interpolator.Scipy1dInterpolator
interp.ScipyGridInterpolator	= interp.interpolator.ScipyGridInterpolator

This subpackage contains the following class. Actual interpolators are subclasses of AbstractInterpolator and not listed here.

classes	
interp.axes_wrapper.AxesWrapper	axis preprocessor
interp.interpolator.Interpolation	interpolation result
interp.interpolator.AbstractInterpolator	base class for interpolators

Note: Interpolation of N data points,

$$(x_{n1}, \ldots, x_{nd}; y_n)$$

for $n = 1, ..., N$ and d is the dimension of parameter space, i.e., the number of parameters, returns a continuous function f satisfying $f(\boldsymbol{x}_n) = y_n$.

Warning: One should distinguish an interpolation f from fitting functions. An interpolation satisfies $f(\boldsymbol{x}_n) = y_n$ but this does not necessarily hold for fitting functions. Meanwhile, an interpolation is defined patch by patch, so its first or higher derivative can be discontinuous, while usually a fit function is globally defined and class C^∞.

A.3.1. `susy_cross_section.interp.axes_wrapper` module

Axis preprocessor for table interpolation.

This module provides a class `AxesWrapper` for advanced interpolation. Each type of modifiers are provided as two functions: parameters-version and value- version, or one for 'x' and the other for 'y'. The former always returns a tuple of floats because there might be multiple parameters, while the latter returns single float value.

Note: Here we summarize interpolations with modified axes. In axis-modification process, we modify the data points

$$(x_{n1}, \ldots, x_{nd}; y_n),$$

with $d + 1$ functions $w_1, \ldots, w_d$ and w_y into

$$X_{ni} = w_i(x_{ni}), \qquad Y_n = w_\text{y}(y_n)$$

and derive the interpolation function $\bar{f}$ based on $(\boldsymbol{X}_n; Y_n)$. Then, the interpolated result is given by

$$f(\boldsymbol{x}) = w_\text{y}^{-1}\Big(\bar{f}\big(w_1(x_1), \ldots, w_d(x_d)\big)\Big).$$

Type Aliases

  VT (= `float`)
      Type representing elements of data points.

  FT (= `Callable[[VT], VT]`)
      Type for wrapper functions w.

  XT (= `List[VT]`)
      Type for parameters x.

  YT (= `VT`)
      Type for the value y.

class AxesWrapper(*wx, wy, wy_inv=None*)

    Bases: object

    Toolkit to modify the x- and y- axes before interpolation.

    In initialization, one can specify wrapper functions predefined, where one can omit wy_inv argument. The following functions are predefined.

- "identity" (or "id", "linear")
- "log10" (or "log")
- "exp10" (or "exp")

    Variables

- **wx** (*list of* FT) – Wrapper functions (or names) for parameters x.
- **wy** (FT) – Wrapper function for the value y.
- **wy_inv** (FT) – The inverse function of wy.

static identity(*x*)

    Identity function as a wrapper.

static log10(*x*)

    Log function (base 10) as a wrapper.

    Note that this is equivalent to natural-log function as a wrapper.

static exp10(*x*)

    Exp function (base 10) as a wrapper.

    Note that this is equivalent to natural-exp function as a wrapper.

wrapped_x(*xs*)

    Return the parameter values after axes modification.

    Parameters **xs** (XT) – Parameters in the original axes

    Returns *XT* – Parameters in the wrapped axes.

    Note: The argument xs is $(x_1, x_2, \ldots, x_d)$, while the returned value is $(X_1, \ldots, X_d) = (w_1(x_1), \ldots, w_d(x_d))$.

---

wrapped_f(*f_bar, type_check=True*)

    Return interpolating function for original data.

    Return the interpolating function applicable to the original data set, given the interpolating function in the modified axes.

    Parameters

- **f_bar** (function of XT to YT) – The interpolating function in the modified axes.
- **type_check** (*bool*) – To perform type-check or not.

    Returns *function of XT to YT* – The interpolating function in the original axes.

> **Note:** The argument f_bar is $\bar{f}$, which is the interpolation function for $(\boldsymbol{X}_n; Y_n)$, and this method returns the function f, which is
>
> $$f(\boldsymbol{x}) = w_{\mathrm{y}}^{-1}\big(\bar{f}(\boldsymbol{X})\big),$$
>
> where $\boldsymbol{X}$ is given by applying `wrapped_x()` to $\boldsymbol{x}$.

A.3.2. `susy_cross_section.interp.interpolator` module

Interpolators of cross-section data.

Type Aliases

`InterpType (= Callable[[Sequence[float]], float])`
Type representing an interpolation function.

class `Interpolation`(*f0, fp, fm, param_names=None*)
Bases: `object`

An interpolation result for values with uncertainties.

This class handles an interpolation of data points, where each data point is given with uncertainties, but does not handle uncertainties due to interpolation.

In initialization, the interpolation results f0, fp, and fm should be specified as functions accepting a list of float, i.e., f0([x1, ..., xd]) etc. If the argument param_names is also specified, the attribute `param_index` is set, which allows users to call the interpolating functions with keyword arguments.

Parameters

- **`f0`** (*InterpType*) – Interpolating function of the central values.

- **`fp`** (*InterpType*) – Interpolating function of values with positive uncertainty added.

- **`fm`** (*InterpType*) – Interpolating function of values with negative uncertainty subtracted.

- **`param_names`** (*list[str], optional*) – Names of parameters.

Variables **`param_index`** (*dict(str, int)*) – Dictionary to look up parameter's position from a parameter name.

`f0`(**args, **kwargs*)
Return the interpolation result of central value.

The parameters can be specified as arguments, a sequence, or as keyword arguments if `param_index` is set.

Returns *float* – interpolated central value.

Examples

For an interpolation with names "foo", "bar", and "baz", the following calls are equivalent:

- `f0([100, 20, -1])`

- `f0(100, 20, -1)`

- `f0(numpy.array([100, 20, -1]))`

- `f0(100, 20, baz=-1)`

- `f0(foo=100, bar=20, baz=-1)`

- `f0(0, 0, -1, bar=20, foo=100)`

__call__(*args, **kwargs*)
  Function call is alias of f0().

fp(*args, **kwargs*)
  Return the interpolation result of upper-fluctuated value.

  Returns *float* – interpolated result of central value plus positive uncertainty.

fm(*args, **kwargs*)
  Return the interpolation result of downer-fluctuated value.

  Returns *float* – interpolated result of central value minus negative uncertainty.

tuple_at(*args, **kwargs*)
  Return the tuple(central, +unc, -unc) at the point.

  Returns *tuple(float, float, float)* – interpolated central value and positive and negative uncertainties.

unc_p_at(*args, **kwargs*)
  Return the interpolated value of positive uncertainty.

  This is calculated not by interpolating the positive uncertainty table but as a difference of the interpolation result of the central and upper - fluctuated values.

  Returns *float* – interpolated result of positive uncertainty.

> **Warning:** This is not the positive uncertainty of the interpolation because the interpolating uncertainty is not included. The same warning applies for: meth: `unc_m_at`.

unc_m_at(*args, **kwargs*)
  Return the interpolated value of negative uncertainty.

  Returns *float* – interpolated result of negative uncertainty.

class AbstractInterpolator
  Bases: object

  A base class of interpolator for values with uncertainties.

  Actual interpolator should implement _interpolate() method, which accepts a pandas.DataFrame object with one value-column and returns an interpolating function (InterpType).

  interpolate(*table*)
    Perform interpolation for values with uncertainties.

Parameters

- **cross_section_table** (*File*) – A cross-section data table.

- **name** (*str*) – Value name of the table to interpolate.

Returns Interpolation – The interpolation result.

class **Scipy1dInterpolator**(*kind='linear', axes='linear'*)
Bases: susy_cross_section.interp.interpolator.AbstractInterpolator

Interpolator for one-dimensional data based on scipy interpolators.

Parameters

- **kind** (*str*) – Specifies the interpolator types.

  linear uses scipy.interpolate.interp1d (kind="linear"), which performs piece-wise linear interpolation.

  spline uses scipy.interpolate.CubicSpline, which performs cubic-spline interpolation. The natural boundary condition is imposed. This is simple and works well if the grid is even-spaced, but is unstable and not recommended if not even-spaced.

  pchip uses scipy.interpolate.PchipInterpolator. This method is recommended for most cases, especially if monotonic, but not suitable for oscillatory data.

  akima uses scipy.interpolate.Akima1DInterpolator. For oscillatory data this is preferred to Pchip interpolation.

- **axes** (*str*) – Specifies the axes preprocess types.

  linear does no preprocess.

  log uses log-axis for values (y).

  loglinear uses log-axis for parameters (x).

  loglog uses log-axis for parameters and values.

> **Warning:** Users should notice the cons of each interpolator, e.g., "spline" and "akima" methods are worse for the first and last intervals or if the grid is not even-spaced, or "pchip" cannot capture oscillations.

Note: kind also accepts all the options for scipy.interpolate.interp1d, but they except for "cubic" are not recommended for cross-section data. The option "cubic" calls scipy.interpolate.interp1d, but it uses the not-a-knot boundary condition, while "spline" uses the natural condition, which imposes the second derivatives at the both ends to be zero.

Note: Polynomial interpolations (listed below) are not included because they are not suitable for cross-section data. They yield in globally-defined polynomials, but such uniformity is not necessary for our purpose and they suffer from so-called Runge phenomenon. If data is expected to be fit by a polynomial, one may use "linear" with axes="loglog".

- scipy.interpolate.BarycentricInterpolator

- scipy.interpolate.KroghInterpolator

See also:

- MATLAB pchip - MathWorks
- Spline methods comparison

class ScipyGridInterpolator(*kind='linear', axes_wrapper=None*)

Bases: `susy_cross_section.interp.interpolator.AbstractInterpolator`

Interpolator for multi-dimensional structural data.

Parameters

- **kind** (*str*) – Specifies the interpolator types. Spline interpolators can be available only for two-parameter interpolations.

    linear uses `scipy.interpolate.RegularGridInterpolator` with method="linear", which linearly interpolates the grid mesh.

    spline alias of "spline33".

    spline33 uses `scipy.interpolate.RectBivariateSpline` with order (3, 3); the numbers may be 1 to 5, but "spline11" is equivalent to "linear".

- **axes_wrapper** (*AxesWrapper, optional*) – Object for axes preprocess. If unspecified, no preprocess is performed.

A.4. **susy_cross_section.utility module**

Utility functions and classes.

Unit	describing a physical unit.
value_format	give human-friendly string representation of values.
get_paths	parse and give paths to data and info files

class Unit(**args*)

Bases: `object`

A class to handle units of physical values.

This class handles units associated to physical values. Units can be multiplied, inverted, or converted. A new instance is equivalent to the product of *args; each argument can be a str, a Unit, or a float (as a numerical factor).

Parameters ***args** (*float, str, or Unit*) – Factors of the new instance.

definitions = {'': [1], '%': [0.01], 'pb': [1000, 'fb']}

The replacement rules of units.

This dictionary defines the replacement rules for unit conversion. Each key should be replaced with the product of its values.

Type *dict[str, list of (float or str)]*

inverse()

Return an inverted unit.

Returns *Unit* – The inverted unit of self.

__imul__(*other*)
Multiply by another unit.

Parameters **other** (*float*, *str*, *or* *Unit*) – Another unit as a multiplier.

__mul__(*other*)
Return products of two units.

Parameters **other** (*float*, *str*, *or* *Unit*) – Another unit as a multiplier.

Returns *Unit* – The product.

__truediv__(*other*)
Return division of two units.

Parameters **other** (*float*, *str*, *or* *Unit*) – Another unit as a divider.

Returns *Unit* – The quotient.

__float__()
Evaluate as a float value if this is a dimension-less unit.

Returns *float* – The number corresponding to this dimension-less unit.

Raises `ValueError` – If not dimension-less unit.

value_format(*value, unc_p, unc_m, unit=None, relative=False*)
Return human-friendly text of an uncertainty-accompanied value.

Parameters

- **value** (*float*) – Central value.

- **unc_p** (*float*) – Positive-direction absolute uncertainty.

- **unc_m** (*float*) – Negative-direction absolute uncertainty.

- **unit** (*str*, *optional*) – Unit of the value and the uncertainties.

- **relative** (*bool*) – Whether to show the uncertainties in relative.

Returns *str* – Formatted string describing the given value.

get_paths(*data_name, info_path=None*)
Return paths to data file and info file.

Parameters

- **data_name** (*pathlib.Path or str*) – Path to grid-data file or a table name predefined in configuration.

  If a file with data_name is found, the file is used. Otherwise, data_name must be a pre-defined table key, or raises KeyError.

- **info_path** (*pathlib.Path or str*, *optional*) – Path to info file, which overrides the default setting.

  The default setting is the grid-file path with suffix changed to ".info".

Returns *Tuple[pathlib.Path, pathlib.Path]* – Paths to data file and info file; absolute if preconfigured.

Raises `FileNotFoundError` – If one of the specified files is not found.

class TypeCheck
Bases: `object`

Singleton class for methods to type assertion.

static is_list(*obj, element_type=None*)
    Return if obj is a list with elements of specified type.

    Parameters

- **obj** – object to test.

- **element_type** (*type or list of type, optional*) – Allowed types for elements.

    Returns *bool* – Validation result.

static is_dict(*obj, key_type=None, value_type=None*)
    Return if obj is a dict with keys/values of specified type.

    Parameters

- **obj** – object to test.

- **key_type** (*type or list of type, optional*) – Allowed types for keys.

- **key_type** – Allowed types for values.

    Returns *bool* – Validation result.

A.5. susy_cross_section.config module

Configuration data of this package.

table_names = {name: data_path or Tuple[data_path, info_path], ...}
    Preset table names and paths to files.

    A *dict* object, where the values show the paths to table and info files. Values are a tuple (`table_file_path, info_file_path`), or `table_file_path` if info_file_path is given by replacing the extension of table_file_path to .info. The path is calculated relative to `table_dir`.

    Type *dict[str, str or tuple[str, str]]*

table_dir = 'susy_cross_section/data'
    Base directory for table data, relative to the package directory.

    Type *str*

package_dir = PosixPath('/Users/misho/.ghq/github.com/misho104/susy_cross_section')
    The package diretory, usually no need to change.

    Type *pathlib.Path*

parse_table_value(*obj*)
    Parse the table values, which might be str or tuple, to a tuple.

    Parameters **obj** (*str or tuple[str, str]*) – The value of `table_names`.

    Returns *tuple[str, str or None]* – The path to grid file and (if specified) to info file.

table_paths(*key, absolute=False*)
    Return the relative paths to table file and info file.

    Parameters

- **key** (*str*) – The key of cross-section table.

- **absolute** (*bool*) – Whether to return absolute paths or not.

29

Returns

Tuple[*pathlib.Path, pathlib.Path*] – The relative path to the grid file and the info file.

The path for info file is returned only if it is configured. The paths are calculated relative to the package directory.

A.6. `susy_cross_section.scripts` module

Scripts for user's ease of handling the data.

For details, see the manual or try to execute with `--help` option.

B. Validations

This package also provides tools to validate interpolation methods, i.e., to estimate uncertainties due to interpolation. We here discuss the validation methods we propose, show some of the validation results, and introduce the procedure for users to test their interpolator or their grid data. The validation results are provided in validations/ directory of this package, together with all the codes to generate the results.

We propose and use two methods for validation. One is based on comparison among several interpolator; it is simple and gives a good measure of the error estimation due to interpolation. Another one is what we call "sieve-interpolation" results[*1], which gives strict upper bounds on the error of interpolations.

B.1. Notation

We use the following notations to discuss our validations. This package compiles each row of the grid data to $(\boldsymbol{x}, y, \Delta^+, \Delta^-)$, where $\boldsymbol{x} \in P \subset \mathbb{R}^D$ is a parameter set with D being the number of parameters, y is the central value for cross section $\sigma(\boldsymbol{x})$, and $\Delta^{\pm}$ are the absolute values of positive and negative uncertainties.

We here assume that the grid is "rectilinear"; i.e., the parameter space P is a D-dimensional cube sliced along all the axes with possibly non-uniform grid spacing. In other words, the grid is defined by lists of numbers, $[c_{11}, c_{12}, \dots], \dots, [c_{D1}, c_{D2}, \dots]$, where each list corresponds to one parameter axis[*2], and all the grid points $(c_{1i_1}, c_{2i_2}, \dots, c_{Di_D})$ are provided in the grid table. Thus we can address each grid point by D-integers, $I := (i_1, \dots, i_D)$, which we will mainly use in the following discussion.

Each patch of tessellation can also be addressed by D-integers. In a D-dimensional grid, a patch is formed by 2^D grid points and we use the smallest index among the points to label a tessellation patch. For example, a patch $S_{i,j}$ of a two-dimensional grid is the square surrounded by the points $(\boldsymbol{x}_{i,j}, \boldsymbol{x}_{i,j+1}, \boldsymbol{x}_{i+1,j}, \boldsymbol{x}_{i+1,j+1})$, or more explicitly,

$$S_{i,j} := \left\{ (x_1, x_2) \Big| c_{1,i} \le x_1 \le c_{1,i+1}, c_{2,j} \le x_2 \le c_{2,j+1} \right\}.$$

An interpolator constructs an interpolating function $f : P \to \mathbb{R}$ from a grid table. It is a continuous function (at least in this package) and satisfies $f(\boldsymbol{x}_I) = y_I$. he interpolators of this package also gives interpolations for values with uncertainties $f^{\pm}$, which satisfy $f^{\pm}(\boldsymbol{x}_I) = y_I \pm \Delta^{\pm}$.

[*1] Or tell me another better name, if you have.

[*2] This array is equivalent to the `levels` attribute of `MultiIndex` used to describe the grid index of data-frame for $D \ge 2$.

To discuss the difference between two functions f_1 and f_2, we define variations and badnesses. The variation v is defined by the relative difference, while the badness b is the ratio of the difference to the ther uncertainty; i.e.,

$$v(\boldsymbol{x}) = \frac{f_2(\boldsymbol{x}) - f_1(\boldsymbol{x})}{f_1(\boldsymbol{x})}, \qquad b(\boldsymbol{x}) = \frac{f_2(\boldsymbol{x}) - f_1(\boldsymbol{x})}{\min\left[f_1^+(\boldsymbol{x}) - f_1(\boldsymbol{x}), f_1(\boldsymbol{x}) - f_1^-(\boldsymbol{x})\right]}.$$

Though they are not symmetric for f_1 and f_2, the differences are negligible for our discussion.

The badness is a good measure to discuss the interpolation uncertainty. For example, $b = 0.5$ means the interpolation uncertainty is the half of the uncertainty σ stored in the grid table, where we can estimate the total uncertainty by

$$\sigma_{\text{total}} = \sqrt{\sigma^2 + (0.5\sigma)^2} \simeq 1.12\sigma.$$

Accordingly, we can safely ignore the difference between two interpolations if $b < 0.3$ (or $\sigma_{\text{total}} < 1.04\sigma$), while we should consider including interpolation uncertainty if $b > 0.5$.

B.2. "Compare" method

This method estimates the interpolation error by comparing results from several interpolators. For $D = 1$, we have several "kinds" of `Scipy1dInterpolator`, while for $D = 2$, we have "linear" and "spline" options for `ScipyGridInterpolator`. The differences among the interpolators will give us a good estimation of the interpolation error. This method does not suit for $D \geq 3$, but such tables are not yet included in this package.

Two examples of $D = 1$ are shown in Fig. 2a and Fig. 2b, where the linear, cubic-spline, Akima [10], and piecewise cubic Hermite interpolating polynomial (PCHIP) [11] interpolators based on log-log axes are compared to each other. In addition, for data tables provided by the NNLLfast collaboration (i.e., in Fig. 2a), we compare our interpolators with their official interpolator; the results are shown by very tiny black dots, which may be seen as a black line overlapping the other results.

In the upper figures the interpolation results are plotted together with the original grid data shown by black vertical lines. All the results, crossing the central values by definition, are overlapped and undistinguishable. The lower plots show the relative differences between the linear and other interpolators. In Fig. 2a the badness is at most 8.9% and we regard those interpolators are equivalent. Meanwhile, in Fig. 2b the differences are visible. Especially, in the first interval, the linear interpolator gives a result considerably different from the other interpolators. which are all based on spline method and thus consistent, corresponding a badness of 0.91. Since we cannot tell which interpolator is giving the most accurate result, an interpolation uncertainty should be introduced for this interval, for example, by multiplying the other uncertainty by a factor $\sqrt{1 + 0.91^2} \simeq 1.35$.

In general, the interpolation results are less accurate for the "surface" region of the parameter space P, which corresponds to the first and last intervals in one-dimension case. For example, in the cubic-spline method, the functions of the first and last intervals are highly dependent on the boundary conditions. Thus users should be very careful if they apply interpolations to the surface regions in their analysis.

Fig. 3 shows a comparison result for two-dimensional case. The linear and spline interpolators, together with the NNLLfast official interpolator, are compared in the gluino pair-production grid for the 13 TeV LHC. The grid spacing is 100 GeV for both axes. The spline interpolator (`ScipyGridInterpolator` with "spline" option) reproduces the official interpolator, while the linear interpolator has small deviations but the points with $b > 0.3$ is found only in the surface regions ("ms" or "mgl" less than 600 GeV). Accordingly, these interpolators are equivalent for non-surface regions, while attention should always be paid for interpolating results of the surface regions.
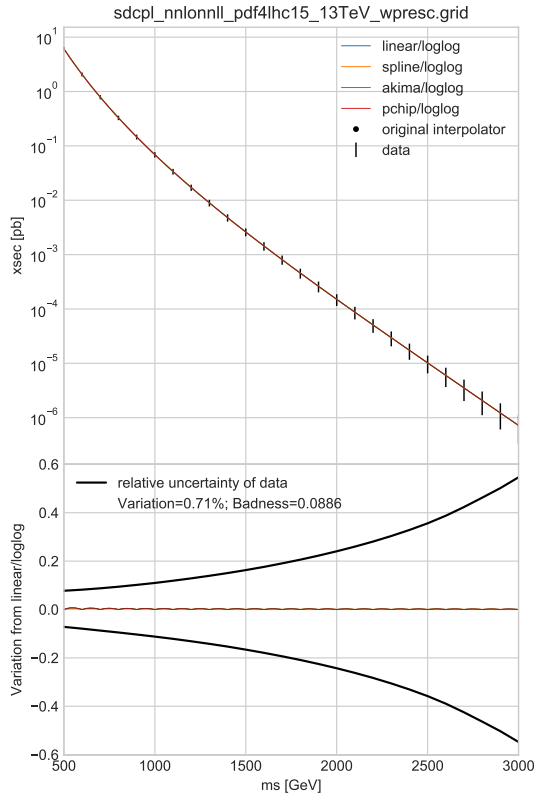
(a) For 13 TeV $pp \rightarrow \tilde{q}\tilde{q}^*$ cross section.
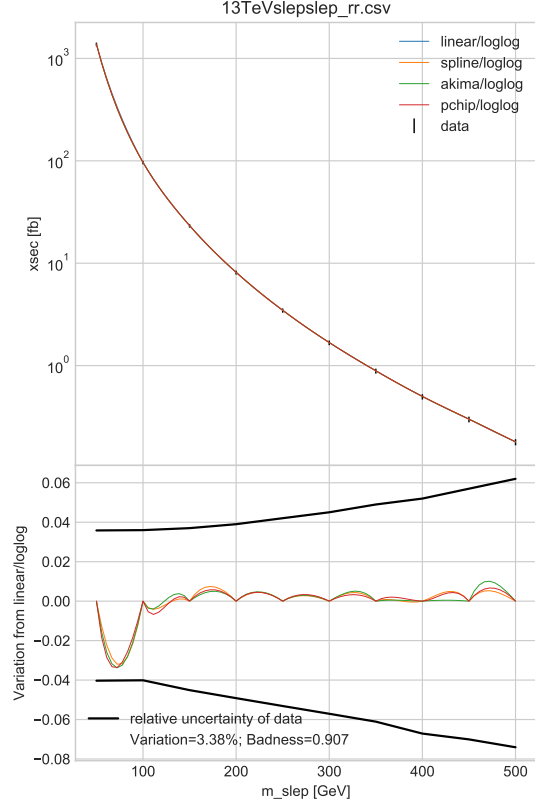
(b) For 13 TeV $pp \rightarrow \tilde{l}_{\mathrm{R}}\tilde{l}_{\mathrm{R}}^*$ cross section.

Figure 2: Validation results of one-dimensional interpolators. The left (right) figure uses the grid table for 13 TeV LHC cross section provided by NNLLfast collaboration (LHC SUSY Cross Section Working Group). In the upper figures the interpolating functions and the original grid data are plotted, where the vertical black lines correspond the uncertainty band of the original data. The lower figures show the relative difference between the linear interpolator and the other interpolators together with the original uncertainty including the scale, PDF, and α_{s} uncertainties. The left figure also includes results from the NNLLfast's official interpolating routine. They are shown by very tiny black dots but exactly overlapping the other lines.

gg_nnlonnll_pdf4lhc15_13TeV_wpresc.grid (1/2)   gg_nnlonnll_pdf4lhc15_13TeV_wpresc.grid (2/2)
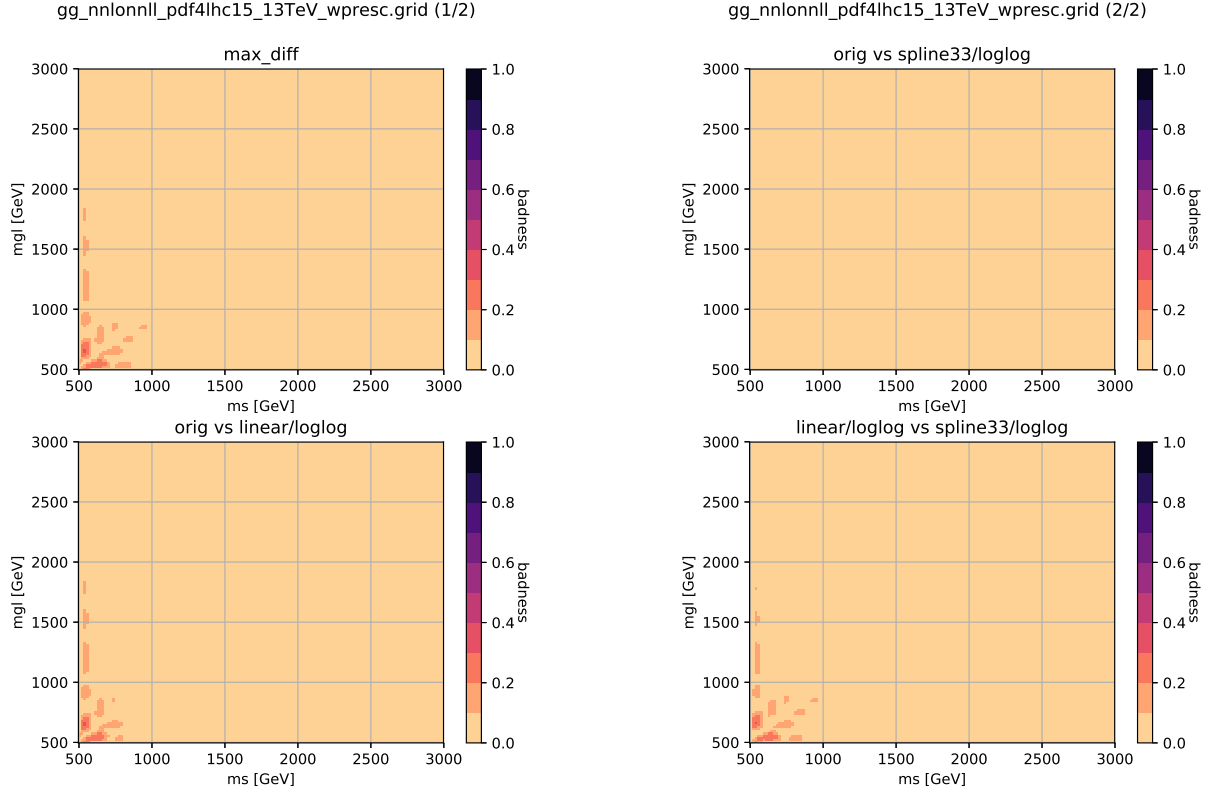
Figure 3: A validation result of two-dimensional interpolators using the 13 TeV $pp \to \tilde{g}\tilde{g}$ cross-section grid provided by NNLLfast collaboration. We here compare the linear and spline interpolators with the official NNLLfast interpolator ("orig") and plot the badness, which is defined by the ratio of the difference to the other uncertainties. The upper-left plot shows the largest difference among the three comparisons, while the respective differences are shown in the other plots.

Validation results for other grid tables are provided together with the package. For most of the grid tables the differences among the interpolators are negligible for non-surface regions, while significant differences are found in surface regions of some grid tables. While users should think of including uncertainties for interpolations in surface regions, one should keep in mind that this "compare" method is just a comparison and thus qualitative rather than quantitative. Thus we put further analyses out of scope of this package and left them to users. For details, see the files in validations/, where codes and instructions to generate the validation results by their own are provided.

B.3. "Sieve" method

This method gives a conservative estimation of the interpolation error by determining the value on a grid point from a "sieved" grid table, which is composed from the original grid table by removing every other line from all the parameter axes.

First, we formally introduce this method. Let us consider a D-dimensional parameter space P that is defined by the grids $[c_{11}, c_{12}, \dots], \dots, [c_{D1}, c_{D2}, \dots]$, as described above. Each grid point is given by

$$\boldsymbol{x}_{i_1, \dots, i_D} = (c_{1i_i}, \dots, c_{Di_D}).$$

We can construct 2^D sieved grid tables, labeled by $P_{r_1 r_2 \cdots r_D}$ with r_n being 0 or 1, as

$$P_{r_1 r_2 \cdots r_D} = \left\{ \boldsymbol{x}_{i_1, \ldots, i_D} \middle| i_n \equiv r_n \bmod 2 \right\}.$$

We can do interpolation over the sieved tables, but the resulting functions $f_{r_1 r_2 \cdots r_D}$ should give much worse results than the original interpolation f.

Let us consider a grid point $\boldsymbol{x}_{3,4,5}$, where we assume $D = 3$ for simplicity. The point is included only in P_{101}, so $f_{101}(\boldsymbol{x}_{3,4,5})$ gives the true value $y_{3,4,5}$, so does f. Meanwhile, the other interpolations f_{ijk} do not give the correct value. In particular, the value is least accurate in f_{010} because all the neighboring points are not included in f_{010}; the point $\boldsymbol{x}_{3,4,5}$ locates the central region of a patch. Therefore, the difference from the true value, $\delta_{3,4,5} := f_{010}(\boldsymbol{x}_{3,4,5}) - y_{3,4,5}$, gives a good estimation of the interpolation error of f_{010} for the region close to $\boldsymbol{x}_{3,4,5}$. We use this difference and resulting variation and badnesses as the error estimation of the original interpolation f around $\boldsymbol{x}_{3,4,5}$.

In summary, the "sieve" method gives a very conservative estimation of the interpolation error around $\boldsymbol{x}_{i_1, \ldots, i_D}$ for an interpolation f by

$$\delta_{i_1, \ldots, i_D} := f_{\bar{r}_1 \cdots \bar{r}_D}(\boldsymbol{x}_{i_1, \ldots, i_D}) - y_{i_1, \ldots, i_D},$$

where $\bar{r}_n := (i_n + 1) \bmod 2$.

Note not only that this method does not give an estimation for the surface points, but also that, for spline-based interpolators, the estimation given by this method becomes extremely conservative for the next-to-surface and next-to-next-to-surface points because they locate at the center of a surface patch in the sieved grid table. Therefore, for spline-based interpolation, it will be ideal to prepare grid points with margins of two or three grid points beyond the region of interest.

Sample results are shown in Fig. 4a, Fig. 4b, and Fig. 5, which are for the same grid tables as in the previous section. In the first figure, $D = 1$ and thus two lines are shown for each interpolator. Those eight lines are overlapped and barely distinguishable; in the lower plot one may see two lines for the linear interpolator, which have a zig-zag form because each interpolator gives the true value at every other points. Since the maximal badness is 0.30, the interpolation error is negligible for this case.

A worse result is shown in Fig. 4b, which indicates that we could not trust the interpolation for $m_{\text{slep}} \lesssim 125 \, \text{GeV}$ if the grid spacing were doubled (100 GeV). Practically, one will avoid using the interpolation, consider to insert additional grid points, or include uncertainty due to interpolation based on the results of the "compare" method for this region. Conversely, we can safely ignore the interpolation error for $m_{\text{slep}} \gtrsim 125 \, \text{GeV}$, which agrees with the "compare" method.

Fig. 5 is an example with $D = 2$; the left (right) figure is a validation for the linear (cubic-spline) interpolator. The upper plots include the sieved interpolation results $f_{\bar{r}_1 \bar{r}_2}(\boldsymbol{x}_{i_1, i_2})$ as the horizontal ticks and the uncertainty range of true values as the vertical lines; by construction, the surface points lacks the sieved interpolation results. The lower figures show the badness for each grid point. Generally, the spline interpolator gives better results, and this is the reason we use the spline interpolato in *get sub-command*. Here, however, one should be careful of the surface region, where the interpolation result is affected by boundary conditions; in fact, the spline interpolator gives much worse results than the linear interpolator at the corner.

Validation results for other grid tables are provided together with the package. In general, as far as avoiding the surface region and the region next to them, the spline interpolator gives very accurate results and users do not have to care of the interpolation error. Meanwhile, the spline interpolation is less trustable for the first and second intervals, where one should consider introducing the interpolation uncertainty. For details, see the files in validations/, where codes and instructions to generate the validation results by their own are provided.

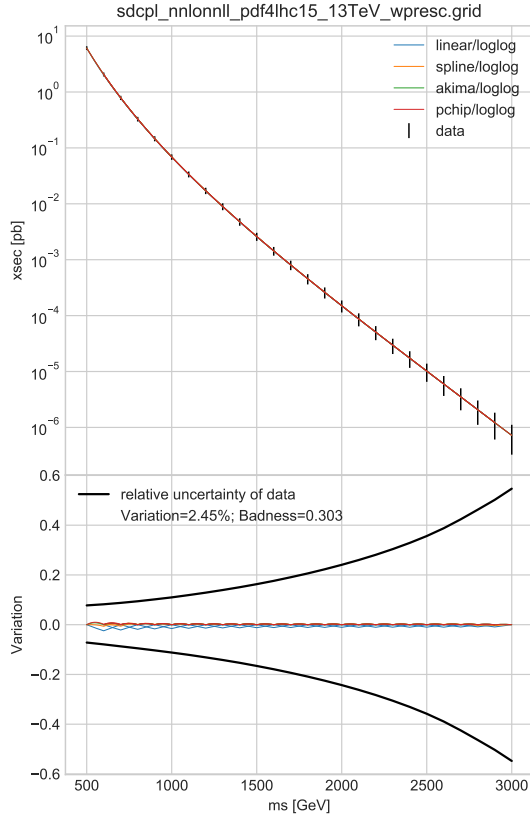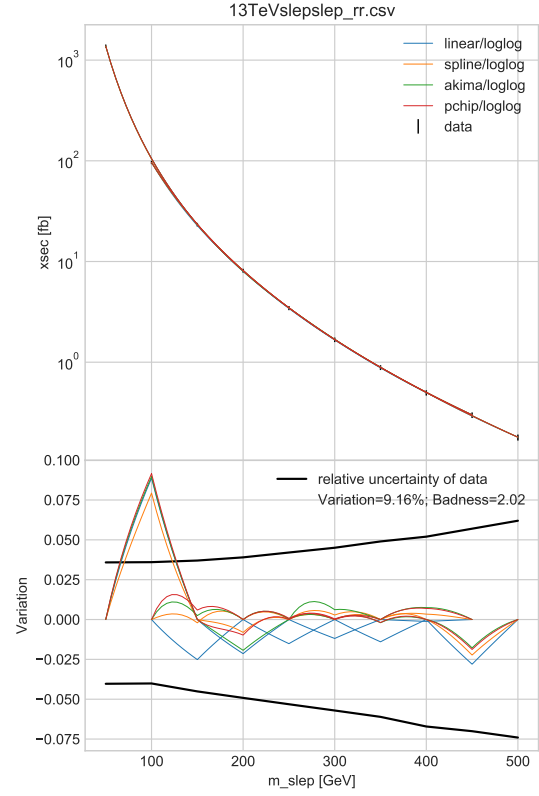(a) For 13 TeV $pp \to \tilde{q}\tilde{q}^*$ cross section.

(b) For 13 TeV $pp \to \tilde{l}_{\mathrm{R}}\tilde{l}_{\mathrm{R}}^*$ cross section.

Figure 4: The same as Fig. 2b but based on "sieve" method. In the upper figures the sieved interpolating functions are plotted with the original grid data, while the lower figures show the variations and badnesses of the sieved interpolating functions.
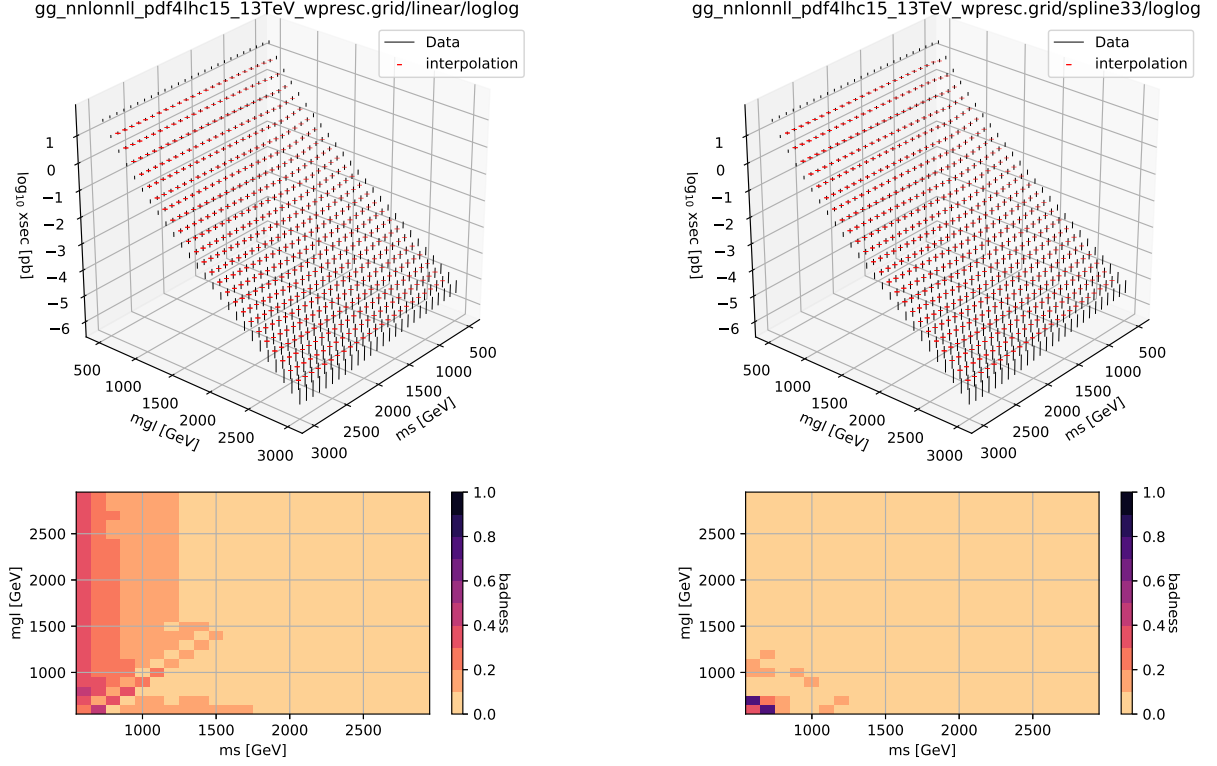
Figure 5: A validation result of two-dimensional interpolators using the 13 TeV $pp \to \tilde{g}\tilde{g}$ cross-section grid provided by NNLLfast collaboration. The left (right) figure is for the linear (cubic-spline) interpolator. In the upper plots the vertical intervals show the uncertainty range of true values and the horizontal ticks show the sieved interpolation results. The lower plots show the badnesses around each of the grid points.

References

[1] **ATLAS** Collaboration, *Luminosity determination in pp collisions at $\sqrt{s} = 8$ TeV using the ATLAS detector at the LHC*, Eur. Phys. J. **C76** (2016) 653 [arXiv:1608.03953].

[2] **CMS** Collaboration, *CMS luminosity measurement for the 2017 data-taking period at* $\sqrt{s} = 13$ TeV, CMS–PAS–LUM–17–004, CERN, 2018.

[3] W. Beenakker, R. Hopker, and M. Spira, *PROSPINO: A Program for the production of supersymmetric particles in next-to-leading order QCD*. hep-ph/9611232.

[4] *Prospino2*. https://www.thphys.uni-heidelberg.de/~plehn/index.php?show=prospino.

[5] B. Fuks, M. Klasen, D. R. Lamprea, and M. Rothering, *Precision predictions for electroweak superpartner production at hadron colliders with Resummino*, Eur. Phys. J. C **73** (2013) 2480 [arXiv:1304.0790].

[6] W. Beenakker, C. Borschensky, M. Krämer, A. Kulesza, and E. Laenen, *NNLL-fast: predictions for coloured supersymmetric particle production at the LHC with threshold and Coulomb resummation*, JHEP **12** (2016) 133 [arXiv:1607.07741].

[7] S. Otten, *et al.*, *DeepXS: Fast approximation of MSSM electroweak cross sections at NLO*. arXiv:1810.08312.

[8] *LHC SUSY Cross Section Working Group*. https://twiki.cern.ch/twiki/bin/view/LHCPhysics/SUSYCrossSections.

[9] ECMA International, *The JSON Data Interchange Format*, Standard ECMA–404 2nd Edition, ECMA International, 2017.

[10] H. Akima, *A New Method of Interpolation and Smooth Curve Fitting Based on Local Procedures*, Journal of the ACM **17** (1970) 589–602.

[11] F. N. Fritsch and R. E. Carlson, *Monotone piecewise cubic interpolation*, SIAM Journal on Numerical Analysis **17** (1980) 238–246.

Python Module Index

Index