SModelS Manual

Release 2.1.0

Gael Alguero, Jan Heisig, Charanjit K. Khosa, Sabine Kraml, Suchita Kulkarni,

Jun 21, 2021

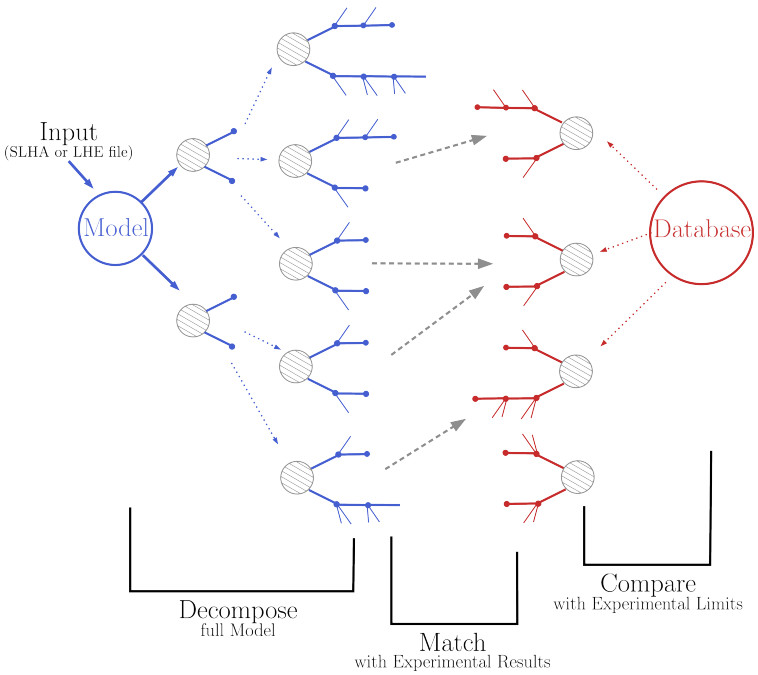
Contents

These pages constitute the SModelS manual.

SModelS is an automatic, public tool for interpreting simplified-model results from the LHC. It is based on a general procedure to decompose Beyond the Standard Model (BSM) collider signatures presenting a Z_2-like symmetry into Simplified Model Spectrum (SMS) topologies. Our method provides a way to cast BSM predictions for the LHC in a model independent framework, which can be directly confronted with the relevant experimental constraints. The main SModelS ingredients are

- the decomposition of the BSM spectrum into SMS topologies
- a database of experimental SMS results
- the interface between decomposition and results database to compute limits

as illustrated in the graphics below.



1 Contents

1.1 What's New

The major novelties of all releases since v1.0 are as follows:

New in Version 2.1.0:

- Ability to merge *Databases* using '+' as a delimiter: "latest_fastlim" and "official_fastlim" are now written as "latest+fastlim", and "official+fastlim".
- useSuperseded flag in getExpResults is marked as deprecated, as we now just put superseded results in separate database
- datasets now have an .isCombinableWith function
- slightly extended output of *summary printer*
- added scan summary (*summary.txt*) when running over multiple files
- added option to slha-printer (*expandedOutput*)
- improved *interactive plots*

New in Version 2.0.0:

- Introduction of *particle class*
- Introduction of model class (see *Basic Input*)
- Input model can now be defined by an SLHA file with *QNUMBERS blocks*
- Unified treatment of SLHA and LHE input files (see *decomposer* and *LHE-reader*)

- *Decomposition* and *Experimental Results* can now handle *lifetime dependent results*
- Added *field "type"* to the experimental results in the database
- Added (optional) *field "intermediateState"* to the experimental results in the database
- Inclusive branches can now describe inclusive vertices
- Added possibility for analysis specific detector size
- New *missing topologies* algorithm and output
- Added "latest" and "latest_fastlim" *Database* abbreviations
- Added support for central database server
- Small bug fix in *likelihood computation*
- Small fix due to an API change in pyhf 0.6
- Changes in output: *width values added*, *coverage groups* and others (see *output description* for details)
- Added option for signal strength multipliers in *cross section calculator*
- Small bug fixes in *models*

New in Version 1.2.4:

- added pyhf support
- pickle path bug fix
- bug fix for parallel xseccomputers
- Introduced the SMODELS_CACHEDIR environment variable to allow for a different location of the cached database file
- fixed dataId bug in datasets

New in Version 1.2.3:

- *database* updated with results from more than 20 new analyses
- server for databases is now smodels.github.io, not smodels.hephy.at
- small bug fix for displaced topologies
- small fix in slha printer, r_expected was r_observed
- *Downloaded database files* now stored in $HOME/.cache/smodels

New in Version 1.2.2:

- Updated official *database*, added T3GQ eff maps and a few ATLAS 13 TeV results, see github database release page
- Database "official" now refers to a database without fastlim results, "official_fastlim", to the official database *with* fastlim
- List displaced signatures in *missing topologies*
- Improved description about lifetime reweighting in doc
- Fix in *cluster* for asymmetric masses

- Small improvements in the *interactive plots tool*

New in Version 1.2.1:

- Fix in particleNames.py for non-MSSM models
- Fixed the marginalize recipe
- Fixed the T2bbWWoff 44 signal regions plots in ConfrontPredictions in manual

New in Version 1.2.0:

- Decomposition and experimental results can include non-MET BSM final states (e.g. heavy stable charged particles)
- Added lifetime reweighting at *decomposition* for meta-stable particles
- Added finalState property for Elements
- Introduction of *inclusive simplified models*
- Inclusion of HSCP and R-hadron results in the database

New in Version 1.1.3:

- Support for *covariance matrices* and combination of signal regions (see *combineSR* in *parameters file*)
- New plotting tool added to smodelsTools (see *Interactive Plots Maker*)
- Path to particles.py can now be specified in parameters.ini file (see *model* in *parameters file*)
- Wildcards allowed when selecting analyses, datasets, txnames (see *analyses*, *txnames* and *dataselector* in *parameters file*)
- Option to show individual contribution from topologies to total theory prediction (see *addTxWeights* in *parameters file*)
- URLs are allowed as database paths (see *path* in *parameters file*)
- Python default changed from python2 to python3
- Fixed lastUpdate bug, now giving correct date
- Changes in pickling (e.g. subpickling, removing redundant zeroes)
- Added fixpermissions to smodelsTools.py, for system-wide installs (see *Files Permissions Fixer*)
- Fixed small issue with pair production of even particles
- Moved the *code documentation* to the manual
- Added *option for installing* within the source folder

New in Version 1.1.2:

- Database update only, the code is the same as v1.1.1

New in Version 1.1.1:

- *C++ Interface*
- Support for pythia8 (see *Cross Section Calculator*)
- improved binary database
- automated SLHA and LHE file detection
- Fix and improvements for missing topologies
- Added SLHA-type output
- Small improvements in interpolation and clustering

New in Version 1.1.0:

- the inclusion of efficiency maps (see *EM-type results*)
- a new and more flexible database format (see *Database structure*)
- inclusion of likelihood and χ^2 calculation for *EM-type results* (see *likelihood calculation*)
- extended information on the *topology coverage*
- inclusion of a database broswer tool for easy access to the information stored in the database (see *database browser*)
- the database now supports also a more efficient *binary format*
- performance improvement for the *decomposition* of the input model
- inclusion of new simplified results to the *database* (including a few 13 TeV results)
- *Fastlim* efficiency maps can now also be used in SModelS

1.2 Installation and Deployment

Standard Installation

SModelS is a Python library that requires Python version 2.6 or later, including version 3, which is the default. It depends on the following *external* Python libraries:

- docutils>=0.3
- numpy>=1.13.0
- scipy>=1.0.0
- unum>=4.0.0
- requests>=2.0.0
- pyslha>=3.1.0
- pyhf>=0.6.1
- jsonpatch>=1.25
- jsonschema>=3.2.0

For speed reasons, we moreover recommend pytorch>=1.8.0 as backend for pyhf. This is, however, optional: if pytorch is not available, SModelS will use the default backend.

In addition, the *cross section computer* provided by *smodelsTools.py* requires:

- Pythia 8.2 (requires a C++ compiler) or Pythia 6.4.27 (requires gfortran)
- NLL-fast 1.2 (7 TeV), 2.1 (8 TeV), and 3.1 (13 TeV) (requires a fortran compiler)

These tools need not be installed separately, as the SModelS build system takes care of that. The current default is that both Pythia6 and Pythia8 are installed together with NLLfast. Finally, the *database browser* provided by *smodelsTools.py* requires IPython, while the *interactive plotter* requires plotly and pandas.

Installation Methods

- The first installation method installs SModelS in the source directory. After downloading the source from the SModelS releases page and extracting it, run:

```
make smodels
```

  in the top-level directory. The installation will remove redundant folders, install the required dependencies (using pip install) and compile Pythia and NLL-fast. If the cross section computer is not needed, one can replace *smodels* with *smodels_noexternaltools* in the above command. In case the Python libraries can not be successfully installed, the user can install them separately using his/her preferred method. Pythia and NLL-fast can also be compiled separately running **make externaltools**.

- If Python's *setuptools* is installed in your machine, SModelS and its dependencies can also be installed without the use of pip. After downloading the source from the SModelS releases page and extracting it, run:

```
setup.py install
```

  within the main smodels directory. If the python libraries are installed in a system folder (as is the default behavior), it will be necessary to run the install command with superuser privilege. Alternatively, one can run setup.py with the "–user" flag:

```
setup.py install --user
```

  If *setuptools* is not installed, you can try to install the external libraries manually and then rerun setup.py. For Ubuntu, SL6 machines and other platforms, a recipe is given below.

  Note that this installation method will install smodels into the default system or user directory (e.g. ~/.local/lib/python3/site-packages/). Depending on your platform, the environment variables $PATH, $PYTHONPATH, $LD_LIBRARY_PATH (or $DYLD_LIBRARY_PATH) might have to be set appropriately.

- Finally, if *pip3* (or *pip*) is installed in your machine, it is also possible to install SModelS directly without the need for downloading the source code:

```
pip3 install smodels
```

  in case of system-wide installs or :

```
pip3 install --user smodels
```

  for user-specific installations.

  Note that this installation method will install smodels into the default system or user directory (e.g. ~/.local/lib/python3/site-packages/). Depending on your platform, the environment variables $PATH, $PYTHONPATH, $LD_LIBRARY_PATH (or $DYLD_LIBRARY_PATH) might have to be set appropriately.

Be aware that the example files and the *parameters file* discussed in the manual will also be located in your default system or user directory. Furthermore the database folder is not included (see *database installation* below).

There is also a diagnostic tool available:

```
smodelsTools.py toolbox
```

should list and check all internal tools (Pythia and NLL-fast) and external (numpy, scipy, unum, ...) dependencies.

In case everything fails, please contact smodels-users@lists.oeaw.ac.at

Installing the SModelS Database

The installation methods explained above (except for pip install) also install SModelS' *database of experimental results* in the smodels-database subdirectory. The first time SModelS is run, a *binary file* will be built using this text database folder, which can then be used in all subsequent runs. However, from v1.1.3 onwards it is recommended to provide the URL of the official database as the database path when running SModelS (see *path* in *parameters file*). In this case the corresponding database version binary file will be automatically downloaded and used. The available database URLs can be found in the SModelS Database releases page .

The complete list of analyses and results included in the database can be consulted at https://smodels.github.io/wiki/ListOfAnalyses. We note that all the results in the official database release have been carefully validated and the validation material can be found at https://smodels.github.io/wiki/Validation.

The database can conveniently be updated independently from SModelS code updates. It suffices to unpack any new database tarball and replace the database directory or provide the *path* to the new folder, binary or URL address. In the same fashion, one can easily add additional results as explained below.

Adding FastLim data

The official SModelS database can be augmented with data from the fastlim results. For using SModelS with the text database, a tarball with the *properly converted* fastlim-1.0 efficiency maps can be found in the smodels-database folder. The tarball then needs to be exploded in the top level directory of the database:

```
cd <smodels-database folder>
tar -xzvf smodels-v1.1-fastlim-1.0.tgz
rm smodels-v1.1-fastlim-1.0.tgz
```

Once the fastlim folders have been added to the database, SModelS auto-detects fastlim results and issues an acknowledgement.

As discussed above, from v1.1.3 onwards it is also possible to directly download the database binary file using the URLs provided in the SModelS Database releases page . Separate URLs are provided for the database including the Fastlim maps, so the user can choose which database to use.

When using the Fastlim results, please properly cite the fastlim paper; for convenience, a bibtex file is provided in the smodels-fastlim tarball.

Finally we point out that when converting the Fastlim efficiency maps efficiencies with a relative statistical uncertainty greater than 25% were set to zero. Also, per default we discard zeroes-only results.

Adding one's own results

The *Database of Experimental Results* is organized as files in an ordinary directory hierarchy. Therefore, adding additional experimental results is a matter of copying and editing text files. Once the new folders and files have been

added following the *database structure format*, SModelS automatically rebuilds the binary (Pickle) database file. The added results will then be available for using with the the SModelS tools.

System-specific Installation Instructions

Installation on Ubuntu >= 16.04

Installation on Ubuntu machines should be straightforward with superuser privileges (if you do not have superuser privileges see instructions below):

- sudo apt install gfortran python-setuptools python-scipy python-numpy python-docutils python-argparse
- setup.py install

Note that the last command can be run as superuser, or with the "–user" flag.

Installation on SL7

Installation on an SL7 or CentOS7 is straightforward:

- yum install gcc-c++ scipy numpy
- pip install unum pyslha argparse

Installation on SL6

Installation on an SL6 (Scientific Linux 6 or Scientific Linux CERN 6) machine is tricky, because SModelS requires a more recent version of *scipy* than is provided by SL6. We succeeded to install SModelS on SL6 by doing:

- yum install gcc-c++ libstdc++-devel libevent-devel python-devel lapack lapack-devel blas blas-devel libgfortran python-distutils-extra

followed by:

- pip install nose unum argparse numpy pyslha scipy

Note, that these steps can safely be done within a Python `virtualenv`. Pip can also be called with the "–user" flag.

Installation on SL5 and similar distributions

In some distributions like SL5, the Python default version may be smaller than 2.6. In these cases, `virtualenv` has to be set up for a Python version >= 2.6. E.g. for Python 2.6, do `virtualenv --python=python2.6 <envname>`, and modify by hand the first line in the executable from `#!/usr/bin/env python3` to `#!/usr/bin/env python2.6`. Then perform the steps listed under `Installation on SL6`.

Installation on other platforms or without superuser privileges using Anaconda

Another easy and platform independent way of installing SModelS without superuser priviledges is via Anaconda (https://www.continuum.io/downloads). Anaconda provides a local installation of pip as well as several additional python packages. Here we assume a version of gfortran is already installed in your system.

- download and install Anaconda for Python 3.6 (https://www.continuum.io/downloads)
- make sure Anaconda's bin and lib folders are added to your system and Python paths

```
PATH="<anaconda-folder>/bin:$PATH"
PYTHONPATH=$PYTHONPATH:"<anaconda-folder>/lib/python3.6/site-packages"
```

and then install SModelS as a user:

```
setup.py install --user
```

In order to make sure all libraries have been correctly installed, you can run:

```
smodelsTools.py toolBox
```

Installation of the C++ interface

From version 1.1.1 on, SModelS comes with a simple C++ interface, see the cpp directory. Obviously, a C++ compiler is need, alongside with the python developers (header) files (libpython-dev on ubuntu, python-devel on rpm-based distros).

1.3 Using SModelS

SModelS can take SLHA or LHE files as input (see *Basic Input*). It ships with a command-line tool *runSModelS.py*, which reports on the SMS *decomposition* and *theory predictions* in several *output formats*.

For users more familiar with Python and the SModelS basics, an example code *Example.py* is provided showing how to access the main SModelS functionalities: *decomposition*, the *database* and *computation of theory predictions*.

The command-line tool (*runSModelS.py*) and the example Python code (*Example.py*) are described below.

Note: For non-MSSM (incl. non-SUSY) input models the user needs to write their own *model.py* file and specify which BSM particles are even or odd under the assumed Z_2 (or similar) symmetry (see *adding new particles*). From version 1.2.0 onwards it is also necessary to define the BSM particle quantum numbers in the same file[1].

---

runSModelS.py

runSModelS.py covers several different applications of the SModelS functionality, with the option of turning various features on or off, as well as setting the *basic parameters*. These functionalities include detailed checks of input SLHA files, running the *decomposition*, evaluating the *theory predictions* and comparing them to the experimental limits available in the *database*, determining *missing topologies* and printing the *output* in several available formats.

Starting on v1.1, *runSModelS.py* is equipped with two additional functionalities. First, it can process a folder containing a set of SLHA or LHE file, second, it supports parallelization of this input folder.

The usage of runSModelS is:

> runSModelS.py [-h] -f FILENAME [-p PARAMETERFILE] [-o OUTPUTDIR] [-d] [-t] [-C] [-V] [-c] [-v VERBOSE] [-T TIMEOUT]

***arguments*:**

-h, --help	show this help message and exit

[1] We note that SLHA files including decay tables and cross sections, together with the corresponding *model.py*, can conveniently be generated via the SModelS-micrOMEGAS interface, see arXiv:1606.03834

-f FILENAME, --filename FILENAME name of SLHA or LHE input file or a directory path (required argument). If a directory is given, loop over all files in the directory

-p PARAMETERFILE, --parameterFile PARAMETERFILE name of parameter file, where most options are defined (optional argument). If not set, use all parameters from smodels/etc/parameters_default.ini

-o OUTPUTDIR, --outputDir OUTPUTDIR name of output directory (optional argument). The default folder is: ./results/

-d, --development if set, SModelS will run in development mode and exit if any errors are found.

-t, --force_txt force loading the text database

-C, --colors colored output

-V, --version show program's version number and exit

-c, --run-crashreport parse crash report file and use its contents for a SModelS run. Supply the crash file simply via '– filename myfile.crash'

-v VERBOSE, --verbose VERBOSE sets the verbosity level (debug, info, warning, error). Default value is info.

-T TIMEOUT, --timeout TIMEOUT define a limit on the running time (in secs).If not set, run without a time limit. If a directory is given as input, the timeout will be applied for each individual file.

A typical usage example is:

```
runSModelS.py -f inputFiles/slha/simplyGluino.slha -p parameters.ini -o ./ -v warning
```

The resulting *output* will be generated in the current folder, according to the printer options set in the *parameters file*.

The Parameters File

The basic options and parameters used by *runSModelS.py* are defined in the parameters file. An example parameter file, including all available parameters together with a short description, is stored in `parameters.ini`. If no parameter file is specified, the default parameters stored in `smodels/etc/parameters_default.ini` are used. Below we give more detailed information about each entry in the parameters file.

- *options*: main options for turning SModelS features on or off
- **checkInput** (True/False): if True, *runSModelS.py* will run the *file check tool* on the input file and verify if the input contains all the necessary information.
- **doInvisible** (True/False): turns *invisible compression* on or off during the *decomposition*.
- **doCompress** (True/False): turns *mass compression* on or off during the *decomposition*.
- **computeStatistics** (True/False): turns the likelihood and χ^2 computation on or off (see *likelihood calculation*). If True, the likelihood and χ^2 values are computed for the *EM-type results*.
- **testCoverage** (True/False): set to True to run the *coverage* tool.
- **combineSRs** (True/False): set to True to use, whenever available, covariance matrices to combine signal regions. NB this might take a few secs per point. Set to False to use only the most sensitive signal region (faster!). Available v1.1.3 onwards.
- *particles*: defines the particle content of the BSM model

- **model**: pathname to the Python file that defines the particle content of the BSM model, given either in Unix file notation ("/path/to/model.py") or as Python module path ("path.to.model"). Defaults to *share.models.mssm* which is a standard MSSM. See smodels/share/models folder for more examples. Directory name can be omitted; in that case, the current working directory as well as smodels/share/models are searched for.
- **promptWidth**: total decay width in GeV above which decays are considered prompt, default is 1e-8; available v2.0 onwards.
- **stableWidth**: total decay width in GeV below which particles are considered as (quasi)stable, default is 1e-25; available v2.0 onwards.
- *parameters*: basic parameter values for running SModelS
- **sigmacut** (float): minimum value for an *element* weight (in fb). *Elements* with a weight below sigmacut are neglected during the *decomposition* of SLHA files (see *Minimum Decomposition Weight*). The default value is 0.005 fb. Note that, depending on the input model, the running time may increase considerably if sigmacut is too low, while too large values might eliminate relevant *elements*.
- **minmassgap** (float): maximum value of the mass difference (in GeV) for perfoming *mass compression*. *Only used if doCompress = True*
- **maxcond** (float): maximum allowed value (in the [0,1] interval) for the violation of *upper limit conditions*. A zero value means the conditions are strictly enforced, while 1 means the conditions are never enforced. *Only relevant for printing the output summary*.
- **ncpus** (int): number of CPUs. When processing multiple SLHA/LHE files, SModelS can run in a parallelized fashion, splitting up the input files in equal chunks. *ncpus = -1* parallelizes to as many processes as number of CPU cores of the machine. Default value is 1. Warning: python already parallelizes many tasks internally.
- *database*: allows for selection of a subset of *experimental results* from the *database*
- **path**: the absolute (or relative) path to the *database*. The user can supply either the directory name of the database, or the path to the *pickle file*. Also http addresses may be given, e.g. https://smodels.github.io/database/official210. Multiple databases may be specified using '+' as a delimiter. Order matters: results will be overwritten according to the sequence specified. The path "official" refers to the official database of your SModelS version – without fastlim; "official+fastlim" includes fastlim results. In addition, the paths "latest+fastlim" and "latest" refer to the latest databases, with and without fastlim results, respectively. See the github database release page for a list of public database versions.
- **analyses** (list of results): set to ['all'] to use all available results. If a list of *experimental analyses* is given, only these will be used. For instance, setting analyses = CMS-PAS-SUS-13-008,ATLAS-CONF-2013-024 will only use the *experimental results* from CMS-PAS-SUS-13-008 and ATLAS-CONF-2013-024. Wildcards (*, ?, [<list-of-or'ed-letters>]) are expanded in the same way the shell does wildcard expansion for file names. *So analyses = CMS* leads to evaluation of results from the CMS-experiment only, for example. *SUS* selects everything containining SUS, no matter if from CMS or ATLAS. Furthermore selection of analyses can be confined on their centre-of-mass energy with a suffix beginning with a colon and an energy string in unum-style, like :13*TeV. Note that the asterisk behind the colon is not a wildcard. :13, :13TeV and :13 TeV are also understood but discouraged.
- **txnames** (list of topologies): set to ['all'] to use all available simplified model *topologies*. The *topologies* are labeled according to the *txname convention*. If a list of *txnames* are given, only the corresponding *topologies* will be considered. For instance, setting txnames = T2 will only consider *experimental results* for $pp \to \tilde{q}+\tilde{q} \to (jet + \tilde{\chi}_1^0) + (jet + \tilde{\chi}_1^0)$ and the *output* will only contain constraints for this topology. *A list of all topologies and their corresponding txnames can be found* here Wildcards (*, ?, [<list-of-or'ed-letters>]) are expanded in the same way the shell does wildcard expansion for file names. So, for example, txnames = T[12]*bb* picks all txnames beginning with T1 or T2 and containg bb as of the time of writing were: T1bbbb, T1bbbt, T1bbqq, T1bbtt, T2bb, T2bbWW, T2bbWWoff
- **dataselector** (list of datasets): set to ['all'] to use all available *data sets*. If dataselector = upperLimit (efficiencyMap), only *UL-type results* (*EM-type results*) will be used. Furthermore, if a list of signal regions (*data sets*)

is given, only the *experimental results* containing these datasets will be used. For instance, if dataselector = SRA mCT150,SRA mCT200, only these signal regions will be used. Wildcards (*, ?, [<list-of-or'ed-letters>]) are expanded in the same way the shell does wildcard expansion for file names. Wildcard examples are given above.

- **dataTypes** dataType of the analysis (all, efficiencyMap or upperLimit). Can be wildcarded with usual shell wildcards: * ? [<list-of-or'ed-letters>]. Wildcard examples are given above.
- *printer*: main options for the *output* format
- **outputType** (list of outputs): use to list all the output formats to be generated. Available output formats are: summary, stdout, log, python, xml, slha.
- *stdout-printer*: options for the stdout or log printer
- **printDatabase** (True/False): set to True to print the list of selected *experimental results* to stdout.
- **addAnaInfo** (True/False): set to True to include detailed information about the *txnames* tested by each *experimental result*. *Only used if printDatabase=True*.
- **printDecomp** (True/False): set to True to print basic information from the *decomposition* (*topologies*, total weights, ...).
- **addElementInfo** (True/False): set to True to include detailed information about the *elements* generated by the *decomposition*. *Only used if printDecomp=True*.
- **printExtendedResults** (True/False): set to True to print extended information about the *theory predictions*, including the PIDs of the particles contributing to the predicted cross section, their masses and the expected upper limit (if available).
- **addCoverageID** (True/False): set to True to print the list of element IDs contributing to each missing topology (see *coverage*). *Only used if testCoverage = True*. This option should be used along with *addElementInfo = True* so the user can precisely identify which elements were classified as missing.
- *summary-printer*: options for the summary printer
- **expandedSummary** (True/False): set True to include in the summary output all applicable *experimental results*, False for only the strongest one.
- *slha-printer*: options for the SLHA printer
  - **expandedOutput** (True/False): set True to print the full list of results. If False only the most constraining result and excluding results are printed.
- *python-printer*: options for the Python printer
- **addElementList** (True/False): set True to include in the Python output all information about all *elements* generated in the *decomposition*. If set to True the output file can be quite large.
- **addTxWeights** (True/False): set True to print the contribution from individual topologies to each theory prediction. Available v1.1.3 onwards.
- *xml-printer*: options for the xml printer
- **addElementList** (True/False): set True to include in the xml output all information about all *elements* generated in the *decomposition*. If set to True the output file can be quite large.
- **addTxWeights** (True/False): set True to print the contribution from individual topologies to each theory prediction. Available v1.1.3 onwards.

The Output

The results of *runSModelS.py* are printed to the format(s) specified by the **outputType** in the *parameters file*. The following formats are available:

- a human-readable *screen output (stdout)* or *log output*. These are intended to provide detailed information about the *database*, the *decomposition*, the *theory predictions* and the *missing topologies*. The output complexity can be controlled through several options in the *parameters file*. Due to its size, this output is not suitable for storing the results from a large scan, being more appropriate for a single file input.
- a human-readable text file output containing a *summary of the output*. This format contains the main SModelS results: the *theory predictions* and the *missing topologies*. It can be used for a large scan, since the output can be made quite compact, using the options in the *parameters file*.
- a *python dictionary* printed to a file containing information about the *decomposition*, the *theory predictions* and the *missing topologies*. The output can be significantly long, if all options in the *parameters file* are set to True. However this output can be easily imported to a Python enviroment, making it easy to access the desired information. For users familiar with the Python language this is the recommended format.
- a *xml file* containing information about the *decomposition*, the *theory predictions* and the *missing topologies*. The output can be significantly long, if all options are set to True. Due to its broad usage, the xml output can be easily converted to the user's preferred format.
- a *SLHA file* containing information about the *theory predictions* and the *missing topologies*. The output follows a SLHA-type format and contains a summary of the most constraining results and the missed topologies.

In addition, when running over multiple files, a simple text output (summary.txt) is generated with basic information about the results for each input file. A detailed explanation of the information contained in each type of output is given in *SModels Output*.

Example.py

Although *runSModelS.py* provides the main SModelS features with a command line interface, users more familiar with Python and the SModelS language may prefer to write their own main program. A simple example code for this purpose is provided in `examples/Example.py`. Below we go step-by-step through this example code:

- *Import the SModelS modules and methods.* If the example code file is not located in the smodels installation folder, simply add "sys.path.append(<smodels installation path>)" before importing smodels. Set SModelS verbosity level.

```
from smodels.tools import runtime
#Define your model (list of BSM particles)
runtime.modelFile = 'smodels.share.models.mssm'
#runtime.modelFile = 'mssmQNumbers.slha'


from smodels.theory import decomposer
from smodels.tools.physicsUnits import fb, GeV, TeV
from smodels.theory.theoryPrediction import theoryPredictionsFor
from smodels.experiment.databaseObj import Database
from smodels.tools import coverage
from smodels.tools.smodelsLogging import setLogLevel
```

- *Set the path to the database URL.* Specify which *database* to use. It can be the path to the smodels-database folder, the path to a *pickle file* or (starting with v1.1.3) a URL path.

```
from smodels.particlesLoader import BSMList
from smodels.share.models.SMparticles import SMList
```

- *Define the input model*. By default SModelS assumes the MSSM particle content. For using SModelS with a different particle content, the user must define the new particle content and set modelFile to the path of the model file (see **particles:model** in *Parameter File*).

```
def main():
    """
    Main program. Displays basic use case.
```

- *Path to the input file*. Specify the location of the input file. It must be a SLHA or LHE file (see *Basic Input*).

```
    # Path to input file (either a SLHA or LHE file)
#     lhefile = 'inputFiles/lhe/gluino_squarks.lhe'
```

- *Set main options for decomposition*. Specify the values of *sigmacut* and *minmassgap*:

```
    # Set main options for decomposition
    sigmacut = 0.01*fb
```

- *Decompose model*. Depending on the type of input format, choose either the slhaDecomposer.decompose or lheDecomposer.decompose method. The **doCompress** and **doInvisible** options turn the *mass compression* and *invisible compression* on/off.

```

    # Decompose model
    toplist = decomposer.decompose(model, sigmacut, doCompress=True, doInvisible=True,
↪ minmassgap=mingap)
```

- *Access basic information* from decomposition, using the topology list and topology objects:

```

    # Access basic information from decomposition, using the topology list and␣
↪topology objects:
    print("\n Decomposition Results: " )
    print("\t  Total number of topologies: %i " %len(toplist) )
    nel = sum([len(top.elementList) for top in toplist])
    print("\t  Total number of elements = %i " %nel )
    #Print information about the m-th topology:
    m = 2
    if len(toplist) > m:
        top = toplist[m]
        print( "\t\t %i-th topology  = " %m,top,"with total cross section =",top.
↪getTotalWeight() )
        #Print information about the n-th element in the m-th topology:
        n = 0
        el = top.elementList[n]
        print( "\t\t %i-th element from %i-th topology  = " %(n,m),el, end="" )
```

output:

```
 Decomposition Results:
          Total number of topologies: 51
          Total number of elements = 14985
                 2-th topology  =  [][2] with total cross section = ['8.00E+00␣
↪[TeV]:3.05E-01 [pb]', '1.30E+01 [TeV]:5.21E-01 [pb]']
```

(continues on next page)

(continued from previous page)

```
                0-th element from 2-th topology  =  [[],[[b,b]]]
                       with final states = ['MET', 'MET']
                       with cross section = ['8.00E+00 [TeV]:2.44E-04 [pb]', '1.
↪30E+01 [TeV]:1.17E-03 [pb]']
                       and masses =  [[6.81E+01 [GeV]], [1.35E+02 [GeV], 6.81E+01␣
↪[GeV]]]
```

- *Load the experimental results* to be used to constrain the input model. Here, all results are used:

```
    # In this case, all results are employed.
```

Alternatively, the getExpResults method can take as arguments specific results to be loaded.

- *Print basic information about the results loaded*. Below we show how to count the number of *UL-type results* and *EM-type results* loaded:

```
    # Count the number of loaded UL and EM experimental results:
    nUL, nEM = 0, 0
    for exp in listOfExpRes:
        expType = exp.datasets[0].dataInfo.dataType
        if expType == 'upperLimit':
            nUL += 1
        elif  expType == 'efficiencyMap':
            nEM += 1
```

output:

```
Loaded Database with 81 UL results and 38 EM results
```

- *Compute the theory predictions* for each *experimental result*. The output is a list of theory prediction objects (for each *experimental result*):

```
    bestResult = None
    for expResult in listOfExpRes:
```

- *Print the results*. For each *experimental result*, loop over the corresponding *theory predictions* and print the relevant information:

```
        print('\n %s ' %expResult.globalInfo.id)
        for theoryPrediction in predictions:
            dataset = theoryPrediction.dataset
            datasetID = theoryPrediction.dataId()
            mass = theoryPrediction.mass
            txnames = [str(txname) for txname in theoryPrediction.txnames]
            PIDs =  theoryPrediction.PIDs
            print("-------------------------" )
            print("Dataset = ",datasetID )   #Analysis name
            print("TxNames = ",txnames )
            print("Prediction Mass = ",mass )   #Value for average cluster mass␣
↪(average mass of the elements in cluster)
            print("Prediction PIDs = ",PIDs )   #Value for average cluster mass␣
↪(average mass of the elements in cluster)
            print("Theory Prediction = ",theoryPrediction.xsection )  #Signal cross␣
↪section
```

output:

```
 ATLAS-SUSY-2015-06
------------------------
Dataset =  SR5j
TxNames =  ['T1', 'T2']
Prediction Mass =  [[5.77E+02 [GeV], 6.81E+01 [GeV]], [5.77E+02 [GeV], 6.81E+01␣
↪[GeV]]]
Prediction PIDs =  [[[1000021, 1000022], [1000021, 1000022]]]
Theory Prediction =  1.30E+01 [TeV]:5.28E-06 [pb]
Condition Violation =  {'None': None}
```

- *Get the corresponding upper limit*. This value can be compared to the *theory prediction* to decide whether a model is excluded or not:

```
            # Get the corresponding upper limit:
```

output:

```
UL for theory prediction =  1.79E+00 [fb]
```

- *Print the r-value*, i.e. the ratio *theory prediction*/upper limit. A value of $r \geq 1$ means that an experimental result excludes the input model. For *EM-type results* also compute the χ^2 and *likelihood*. Determine the most constraining result:

```
            r = theoryPrediction.getRValue()
            print("r = ",r )
            #Compute likelihhod and chi^2 for EM-type results:
            if dataset.getType() == 'efficiencyMap':
                theoryPrediction.computeStatistics()
                print('Chi2, likelihood=', theoryPrediction.chi2, theoryPrediction.
↪likelihood )
            if r > rmax:
                rmax = r
                bestResult = expResult.globalInfo.id
```

output:

```
r =  0.0029506296753791803
Chi2, likelihood= 2.377901422385566 0.007168380743561493
```

- *Print the most constraining experimental result*. Using the largest r-value, determine if the model has been excluded or not by the selected *experimental results*:

```
    # Print the most constraining experimental result
    print("\nThe largest r-value (theory/upper limit ratio) is ",rmax )
    if rmax > 1.:
        print("(The input model is likely excluded by %s)" %bestResult )
```

output:

```
The largest r-value (theory/upper limit ratio) is  1.2039296443268397
(The input model is likely excluded by CMS-SUS-13-006)
```

- *Identify missing topologies*. Using the output from decomposition, identify the *missing topologies* and print some basic information:

```
    #Find out missing topologies for sqrts=8*TeV:
    uncovered = coverage.Uncovered(toplist,sqrts=8.*TeV)
    #First sort coverage groups by label
    groups = sorted(uncovered.groups[:], key = lambda g: g.label)
    #Print uncovered cross-sections:
    for group in groups:
        print("\nTotal cross-section for %s (fb): %10.3E\n" %(group.description,group.
↪getTotalXSec()))

    missingTopos = uncovered.getGroup('missing (prompt)')
    #Print some of the missing topologies:
    if missingTopos.generalElements:
        print('Missing topologies (up to 3):' )
        for genEl in missingTopos.generalElements[:3]:
            print('Element:', genEl)
            print('\tcross-section (fb):', genEl.missingX)
    else:
        print("No missing topologies found\n")

    missingDisplaced = uncovered.getGroup('missing (displaced)')
    #Print elements with displaced decays:
    if missingDisplaced.generalElements:
        print('\nElements with displaced vertices (up to 2):' )
        for genEl in missingDisplaced.generalElements[:2]:
            print('Element:', genEl)
            print('\tcross-section (fb):', genEl.missingX)
    else:
```

output:

```
Total missing topology cross section (fb):  1.408E+04

Total cross section where we are outside the mass grid (fb):  9.481E+02

Total cross section in long cascade decays (fb):  5.969E+03

Total cross section in decays with asymmetric branches (fb):  7.943E+03

Missing topologies (up to 3):
Topology: [[],[]](MET,MET)
Contributing elements (up to 2):
[[],[]] cross-section (fb): 3.8832839999999997
[[],[]] cross-section (fb): 0.557261132427282
Topology: [[],[[W]]](MET,MET)
Contributing elements (up to 2):
[[],[[W+]]] cross-section (fb): 0.32675012450355956
[[],[[W+]]] cross-section (fb): 0.15158171738285436
Topology: [[],[[Z]]](MET,MET)
Contributing elements (up to 2):
[[],[[Z]]] cross-section (fb): 1.894204563779664
[[],[[Z]]] cross-section (fb): 0.17320237949111086

Elements outside the grid (up to 2):
Topology: [[[W]],[[higgs]]](MET,MET)
Contributing elements (up to 4):
[[[W+]],[[higgs]]] cross-section (fb): 0.046455022590588216
```

(continues on next page)

(continued from previous page)

```
        mass: [[2.93E+02 [GeV], 6.81E+01 [GeV]], [2.92E+02 [GeV], 1.35E+02 [GeV]]]
[[[W+]],[[higgs]]] cross-section (fb): 0.062213057626625254
        mass: [[2.93E+02 [GeV], 1.35E+02 [GeV]], [2.66E+02 [GeV], 6.81E+01 [GeV]]]
[[[W+]],[[higgs]]] cross-section (fb): 0.22055328335196533
        mass: [[2.93E+02 [GeV], 1.35E+02 [GeV]], [2.92E+02 [GeV], 6.81E+01 [GeV]]]
[[[W-]],[[higgs]]] cross-section (fb): 0.06242074057131094
        mass: [[2.93E+02 [GeV], 1.35E+02 [GeV]], [2.92E+02 [GeV], 6.81E+01 [GeV]]]
Topology: [[[Z]],[[higgs]]](MET,MET)
Contributing elements (up to 4):
[[[Z]],[[higgs]]] cross-section (fb): 0.36401204399531084
        mass: [[2.66E+02 [GeV], 6.81E+01 [GeV]], [2.92E+02 [GeV], 6.81E+01 [GeV]]]
[[[Z]],[[higgs]]] cross-section (fb): 0.035568533782123934
        mass: [[2.66E+02 [GeV], 6.81E+01 [GeV]], [2.92E+02 [GeV], 1.35E+02 [GeV]]]
[[[Z]],[[higgs]]] cross-section (fb): 0.03328455299337171
        mass: [[2.66E+02 [GeV], 1.35E+02 [GeV]], [2.92E+02 [GeV], 6.81E+01 [GeV]]]
[[[Z]],[[higgs]]] cross-section (fb): 0.024739761811209363
        mass: [[2.92E+02 [GeV], 6.81E+01 [GeV]], [2.66E+02 [GeV], 6.81E+01 [GeV]]]
```

It is worth noting that SModelS does not include any statistical treatment for the results, for instance, correction factors like the "look elsewhere effect". Due to this, the results are claimed to be "likely excluded" in the output.

Notes:

- For an SLHA *input file*, the decays of SM *particles* (or BSM Z_2-even particles) are always ignored during the decomposition. Furthermore, if there are two cross sections at different calculation order (say LO and NLO) for the same process, only the highest order is used.
- The list of *elements* can be extremely long. Try setting **addElementInfo** = False and/or **printDecomp** = False to obtain a smaller output.
- A comment of caution is in order regarding naively using the highest r-value reported by SModelS, as this does not necessarily come from the most sensitive analysis. For a rigorous statistical interpretation, one should use the r-value of the result with the highest *expected* r (r_{exp}). Unfortunately, for *UL-type results*, the expected limits are often not available; r_{exp} is then reported as N/A in the SModelS output.

1.4 SModelS Tools

Inside SModelS there is a number of tools that may be convenient for the user:

- a *cross section calculator* based on Pythia8 (or Pythia6) and NLLfast,
- *SLHA and LHE file checkers* to check your input files for completeness and sanity,
- a *database browser* to provide easy access to the *database* of experimental results,
- a plotting tool to make *interactive plots* based on plotly (v1.1.3 onwards),
- a *file permissions fixer* to fix a problem with file permissions for the cross section computers in system-wide installs, and
- a *toolbox* to quickly show the state of the external tools.

Cross Section Calculator

This tool computes LHC production cross sections for *MSSM particles* and writes them out in *SLHA convention*. This can in particular be convenient for adding cross sections to SLHA input files, see *Basic Input*. The calculation is done at LO with Pythia8 or Pythia6.4 ; K-factors for colored particles are computed with NLLfast. Signal strength multipliers can optionally be supplied for each "mother" particle.

The usage of the cross section calculator is:

smodelsTools.py xseccomputer [-h] [-s SQRTS [SQRTS ...]] [-e NEVENTS] [-v VERBOSITY] [-c NCPUS] [-p] [-P] [-q] [-C] [–noautocompile] [-k] [-6] [-8] [-n] [-N] [-O] [-S SSMULTIPLIERS] -f FILENAME

***arguments*:**

-h, --help show this help message and exit

-s SQRTS, --sqrts SQRTS sqrt(s) TeV. Can supply more than one value (as a space separated list). Default is both 8 and 13.

-e NEVENTS, --nevents NEVENTS number of events to be simulated [10000].

-v VERBOSITY, --verbosity VERBOSITY Verbosity (debug, info, warning, error)

-c NCPUS, --ncpus NCPUS number of cores to be used simultaneously. -1 means 'all'.

-p, --tofile write cross sections to file (only highest order)

-P, --alltofile write all cross sections to file, including lower orders

-q, --query only query if there are cross sections in the file

-C, --colors colored terminal output

--noautocompile turn off automatic compilation

-k, --keep do not unlink temporary directory

-6, --pythia6 use pythia6 for LO cross sections

-8, --pythia8 use pythia8 for LO cross sections (default)

-n, --NLO compute at the NLO level (default is LO)

-N, --NLL compute at the NLO+NLL level (takes precedence over NLO, default is LO)

-O, --LOfromSLHA use LO cross sections from file to compute the NLO or NLL cross sections

-S SSMULTIPLIERS, --ssmultipliers SSMULTIPLIERS Signal strength multipliers, provided as dictionary of pids

-f FILENAME, --filename FILENAME SLHA file to compute cross sections for. If a directory is given, compute cross sections for all files in directory.

Further Pythia parameters are defined in `smodels/etc/pythia8.cfg` (for Pythia 8) or `smodels/etc/pythia.card` (for Pythia 6).

A typical usage example is:

```
smodelsTools.py xseccomputer -s 8 13 -e 10000 -p -f inputFiles/slha/higgsinoStop.slha
```

which will compute 8 TeV and 13 TeV LO cross sections (at the LHC) for all MSSM processes using 10k MC events. If, *after* the LO cross sections have been computed, one wants to add the NLO+NLL cross sections for gluinos and squarks:

```
smodelsTools.py xseccomputer -s 8 13 -p -N -O -f inputFiles/slha/higgsinoStop.slha
```

The resulting file will then contain LO cross sections for all MSSM processes and NLO+NLL cross sections for the available processes in NLLfast (gluino and squark production). When reading the input file, SModelS will then use only the highest order cross sections available for each process.

An example using signal strength multipliers (*available from SModelS v2.0 onwards*) is:

```
smodelsTools.py xseccomputer -s 8 13 -e 10000 --ssmultipliers "{ (1000021,1000021): 4.
↪0, (1000001,-10000001): 2.0 }" -p -f inputFiles/slha/higgsinoStop.slha
```

This will compute 8 TeV and 13 TeV LO cross sections as above, but the cross section for gluino-pair production (pid 1000021) gets enhanced by a factor of 4, and squark-antisquark production gets enhanced by a factor of 2. For the pids, strings can be supplied instead of integers. Unix filename wildcard syntax is also supported. E.g. '100000?' matches all left-handed squarks but no anti-squarks, '*1000001' matches both (left-handed) down and anti-down. Multiple signal strength multipliers may be applicable to a single theory prediction.

Note that signal strength multipliers get applied only to LO cross sections. This means they are propagated to NLO and NLL level only if the LO cross sections are computed first and the NLO/NLL corrections added afterwards. In other words, if the xseccomputer is called with -n or -N argument but without -O (–LOfromSLHA), the –ssmultipliers argument will be ignored.

- **The cross section calculation is implemented by the** xsecComputer function

Input File Checks

As discussed in *Basic Input*, SModelS accepts both SLHA and LHE input files. It can be convenient to perform certain sanity checks on these files as described below.

- **The input file checks are implemented by the** FileStatus class

LHE File Checker

For a LHE input file only very basic checks are performed, namely that

- the file exists,
- it contains at least one event,
- the information on the total cross section and the center of mass energy can be found.

The usage of the LHE checker is simply:

smodelsTools.py lhechecker [-h] -f FILENAME

***arguments*:**

-h, --help show this help message and exit

-f FILENAME, --filename FILENAME name of input LHE file

A typical usage example is:

```
smodelsTools.py lhechecker -f inputFiles/lhe/gluino_squarks.lhe
```

SLHA File Checker

The SLHA file checker allows to perform quite rigorous checks of SLHA input files. Concretely, it verifies that

- the file exists and is given in SLHA format,
- the file contains masses and decay branching ratios in standard SLHA format,
- the file contains cross sections according to the *SLHA format for cross sections*,

In addition, one can ask that

- all decays listed in the DECAY block are kinematically allowed, *i.e.* the sum of masses of the decay products may not exceed the mother mass. *This check for "illegal decays" is turned off by default.*

If any of the above tests fail (return a negative result), an error message is shown.

The usage of the SLHA checker is:

smodelsTools.py slhachecker [-h] [-xS] [-illegal] [-dB] -f FILENAME

***arguments*:**

-h, --help	show this help message and exit
-xS, --xsec	turn off the check for xsection blocks
-illegal, --illegal	turn on check for kinematically forbidden decays
-dB, --decayBlocks	turn off the check for missing decay blocks
-f FILENAME, --filename FILENAME	name of input SLHA file

A typical usage example is:

```
smodelsTools.py slhachecker -f inputFiles/slha/gluino_squarks.slha
```

Running this will print the status flag and a message with potential warnings and error messages.

Note: In SModelS versions prior to 1.2, the SLHA file checker also checked for the existence of displaced vertices or heavy stable charged particles in the input file. Since the inclusion of long lived signatures in SModelS, these checks are no longer done by the SLHA file checker.

Database Browser

In several cases the user might be interested in an easy way to directly access the *database* of *Experimental Results*. This can be conveniently done using the database browser. The browser owns several methods to select *Experimental Results* or *DataSets* satisfying some user-defined conditions as well as to access the meta data and data inside each *Experimental Result*.

The usage of the browser interface is:

smodelsTools.py database-browser [-h] -p PATH_TO_DATABASE [-t]

***arguments*:**

-h, --help	show this help message and exit
-p PATH_TO_DATABASE, --path_to_database PATH_TO_DATABASE	path to SModelS database
-t, --text	load text database, dont even search for binary database file

A typical usage example is:

```
smodelsTools.py database-browser -p ./smodels-database
```

Loading the database may take a few seconds if the *binary database file* exists. Otherwise the *pickle file* will be created. Starting the browser opens an IPython session, which can be used to select specific experimental results (or groups of experimental results), check upper limits and/or efficiencies for specific masses/topologies and access all the available information in the database. A simple example is given below:

```
In [1]: print ( browser )  #Print all experimental results in the browser
['ATLAS-SUSY-2015-01', 'ATLAS-SUSY-2015-01', 'ATLAS-SUSY-2015-02', 'ATLAS-SUSY-2015-02
↪', ...

In [2]: browser.selectExpResultsWith(txName = 'T1tttt', dataType = 'upperLimit')
↪#Select only the UL results with the topology T1tttt

In [3]: print ( browser ) #Print all experimental results in the browser (after␣
↪selection)
['ATLAS-SUSY-2015-09', 'CMS-SUS-15-002', 'CMS-PAS-SUS-16-014', 'CMS-PAS-SUS-16-015', .
↪..

In [4]: gluinoMass, LSPmass = 800.*GeV, 100.*GeV  #Define masses for the T1tttt␣
↪topology

In [5]: browser.getULFor('CMS-SUS-15-002','T1tttt',[[gluinoMass,LSPmass],[gluinoMass,
↪LSPmass]]) #Get UL for a specific experimental result
Out[5]: 2.23E+01 [fb]

In [6]: for expResult in browser[:5]:  #Get the upper limits for the first five of␣
↪the selected results for the given topology and mass
   ...:     print ( expResult.getValuesFor('id'),'UL = ',expResult.
↪getUpperLimitFor(txname='T1tttt',mass=[[gluinoMass,LSPmass],[gluinoMass,LSPmass]]) )
   ...:
   ['ATLAS-SUSY-2015-09'] UL =  None
   ['CMS-PAS-SUS-16-014'] UL =  4.10E+01 [fb]
   ['CMS-PAS-SUS-16-015'] UL =  1.80E+01 [fb]
   ['CMS-PAS-SUS-16-016'] UL =  5.76E+01 [fb]
   ['CMS-PAS-SUS-16-019'] UL =  1.37E+01 [fb]


In [7]: for expResult in browser[:5]:  #Print the luminosities for the first five␣
↪selected experimental results
   ...:     print ( expResult.getValuesFor('id'),expResult.getValuesFor('lumi') )
   ...:
   ['ATLAS-SUSY-2015-09'] [3.20E+00 [1/fb]]
   ['CMS-PAS-SUS-16-014'] [1.29E+01 [1/fb]]
   ['CMS-PAS-SUS-16-015'] [1.29E+01 [1/fb]]
   ['CMS-PAS-SUS-16-016'] [1.29E+01 [1/fb]]
   ['CMS-PAS-SUS-16-019'] [1.29E+01 [1/fb]]
```

Further Python example codes using the functionalities of the browser can be found in *Howto's*.

- **The Database browser tool is implemented by the** Browser class

Interactive Plots Maker

This tool allows to easily produce interactive plots which relate the SModelS output with information on the user's model stored in the SLHA files. It gives 2d plots in the parameter space defined by the user, with additional user-defined information appearing in hover boxes. The output is in html format for viewing in a web browser. The aim is not to make publication-ready plots but to facilitate getting an overview of e.g. the properties of points in a scan. NB: this needs SLHA model input and SModelS python output!

Required python packages are: plotly, pandas, pyslha, os, decimal

The usage of the interactive plots tool is:

smodelsTools.py interactive-plots [-h] [-p PARAMETERS] [-m MODELFILE] -f SMODELSFOLDER -s SLHAFOLDER [-o OUTPUTFOLDER] [-N NPOINTS] [-v VERBOSITY]

***arguments*:**

-h, --help show this help message and exit

-p PARAMETERS, --parameters PARAMETERS path to the parameters file [./iplots_parameters.py]

-m MODELFILE, --modelFile MODELFILE path to the model.py file

-f SMODELSFOLDER, --smodelsFolder SMODELSFOLDER path to the smodels folder or tarball (.tar.gz) with the SModelS python output files.

-s SLHAFOLDER, --slhaFolder SLHAFOLDER path to the SLHA folder or tarball (.tar.gz) with the SLHA input files.

-o OUTPUTFOLDER, --outputFolder OUTPUTFOLDER path to the output folder, where the plots will be stored. [./iplots]

-N NPOINTS, --npoints NPOINTS How many (randomly selected) points will be included in the plot. If -1 all points will be read and included (default = -1).

-v VERBOSITY, --verbosity VERBOSITY Verbosity (debug, info, warning, error)

A typical usage example is:

```
smodelsTools.py interactive-plots -f inputFiles/scanExample/smodels-output/ -s␣
↪inputFiles/scanExample/slha -p iplots_parameters.py -o results/iplots/
```

which will produce 3x11 plots in the gluino vs squark mass plane from a small scan example, viewable in a web browser.

iplots parameters file

The options for the interactive plots tool are defined in a parameters file, *iplots_parameters.py* in the above example. An example file, including all available parameters together with a short description, is stored in `iplots_parameters.py`. Since the plotting information is model dependent, there is no default setting – the iplots parameters file is mandatory input. Below we give more detailed information about each entry in this file.

- *plot_title*: main overall title for your plots, typically the model name.
- *x and y axes*: SLHA block and PDG code number of the variables you want to plot, e.g. [‘MASS’, 1000021].
  - **variable_x**: In a list form, give the the block and PDG code number of the x-axis variable, to find it in the SLHA file. Example: variable_x = [‘MASS’, 1000021]. Alternatively, you can provide custom names (instead of extracting them from a model.py file) to your x-axis variable in a dictionary form, e.g. {‘m_gluino’: [‘MASS’, 1000021]}.
  - **variable_y**: same for the y-axis. Example: variable_y = [‘MASS’, 1000022] or variable_y ={‘m_neutralino1’: [‘MASS’, 1000022]}
- *spectrum hover information*: defines which information from the input SLHA file will appear in the hover box. The syntax is again either a list or a dictionary.
  - **slha_hover_information**: information from the input SLHA file, e.g. model parameters or masses. Example: slha_hover_information = [[‘MASS’, 1000021], [‘MASS’, 2000002], [‘MASS’, 1000022]]. Alternatively, slha_hover_information = {‘m_gluino’: [‘MASS’, 1000021], ‘m_suR’: [‘MASS’, 2000002], ‘m_LSP’: [‘MASS’, 1000022]}

  - **ctau_hover_information**: displays the mean decay length in meter for the listed particle(s). Example: ctau_hover_information = [1000024] or = {'ctau_chi1+': 1000024}
  - **BR_hover_information**: defines for which particle(s) to display decay channels and branching ratios. Example: BR_hover_information = [1000021] or = {'BR_gluino': 1000021}. **WARNING:** Lists of branching ratios can be very long, so the may not fit in the hover box. One can define the number of entries with **min_BR**, e.g. min_BR = .05 (default 'all').

- *SModelS hover information*: defines, as a list of keywords, which information to display from the SModelS output. Example: smodels_hover_information = ['SModelS_status', 'r_max', 'Tx', 'Analysis', 'chi2', 'MT_max', 'MT_max_xsec', 'MT_total_xsec', 'MT_outgrid_xsec', 'MT_prompt_xsec', 'MT_displaced_xsec', 'file']. The options are:
  - **SmodelS_status**: prints whether the point is excluded or not by SModelS
  - **r_max**: shows the highest r-value for each parameter point
  - **chi2**: shows the chi^2 value, if available (if not, the output is 'none')
  - **Tx**: shows the topology/ies which give r_max
  - **Analysis**: shows the experimental analysis from which the strongest constraint (r_max) comes from
  - **MT_max**: shows the missing topology with the largest cross section (in SModelS bracket notation)
  - **MT_max_xsec**: shows the cross section of MT_max
  - **MT_total_xsec**: shows the total missing cross section (i.e. the sum of all missing topologies cross sections)
  - **MT_prompt_xsec**: Shows the total cross section from prompt missing topologies
  - **MT_displaced_xsec**: Shows the total cross section from displaced missing topologies
  - **MT_outgrid_xsec**: shows the total missing cross section outside the mass grids of the experimental results
  - **file**: shows the name of the input spectrum file

- *Choice of plots to make*
  - **plot_data**: which points you want to plot; the options are: all, non-excluded, excluded points. Example: plot_data = ['all', 'non-excluded', 'excluded']
  - **plot_list**: which quantities to plot in the x,y plane; the same options as for SModels hover information apply. Example: plot_list = ['SModelS_status', 'r_max', 'Tx', 'Analysis', 'chi2', 'MT_max', 'MT_max_xsec', 'MT_total_xsec', 'MT_outgrid_xsec', 'MT_prompt_xsec', 'MT_displaced_xsec', 'file']

File Permissions Fixer

In case the software was installed under a different user than it is used (as is the case for system-wide installs), we ship a simple tool that fixes the file permissions for the cross section calculation code.

The usage of the permissions fixer is:

smodelsTools.py fixpermissions [-h]

***arguments*:**

-h, --help show this help message and exit

Execute the command as root, i.e.:

```
sudo smodelsTools.py fixpermissions
```

ToolBox

As a quick way to show the status of all external tools, use **the toolbox:**

smodelsTools.py toolbox [-h] [-c] [-l] [-m]

***arguments*:**

-h, --help	show this help message and exit
-c, --colors	turn on terminal colors
-l, --long	long output lines
-m, --make	compile packages if needed

1.5 Detailed Guide to SModelS

Basic Concepts and Definitions

Throughout this manual, several concepts are used extensively. Here we define the most important ones, their respective nomenclature and some useful notation. The concepts related to the basic building blocks of the *decomposition* of a full model into a sum of Simplified Models (or *elements*) are described in *Theory Definitions*, while the concepts relevant for the database of experimental results are given in *Database Definitions*.

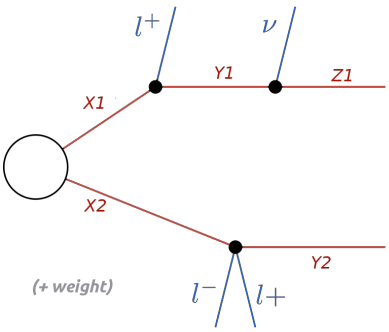
Simplified Model Definitions

The so-called theory module contains the basic tools necessary for decomposing the input model into simplified model *topologies* and using the output of the decomposition to compute the *theoretical prediction* for a given *experimental result*.

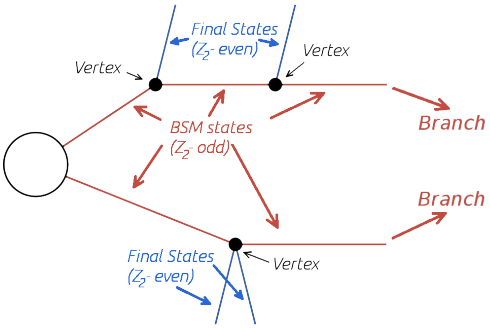
The applicability of SModelS is currently restricted to models which contain a Z_2 symmetry (R-parity in SUSY, K-parity in UED, ...). This is required in order to provide a clear structure for the simplified model topologies appearing during the *decomposition* of the input model. Below we describe the basic concepts and language used in SModelS to describe the simplified model topologies.

Elements

A simplified model topology representing a specific cascade decay of a pair of BSM states produced in the hard scattering is called an element in the SModelS language. Elements contain the Z_2-even particles appearing in the cascade decay and the BSM (Z_2-odd) states which have decayed or appear in the last step of the decay. A representation of an element is shown below:



An element may also hold information about its corresponding weight (cross section times branching ratio times efficiency).[1] The overall properties of an element are illustrated in the scheme below:



SModelS works under the inherent assumption that, for collider purposes, all the essential properties of a BSM model can be encapsulated by its elements. Such an assumption is extremely helpful to cast the theoretical predictions of a specific BSM model in a model-independent framework, which can then be compared against the corresponding experimental limits. From v2.0 onwards elements hold *particles* and two elements will be considered equal if both their topological structure and *particles* are equal. Below we describe in more detail the element properties and their implementation in SModelS.

- **Elements are described by the** Element Class

Particles

The basic building block of simplified model *elements* are particles, which can be both SM (e.g. l^+, l^-, ν in the *figure above*) or BSM states (e.g. $X1, X2, Y1, Y2, Z1$ in the *figure above*). The BSM particles are defined by the input model (see *model* in *parameters file*), while the SM particles are defined in SMparticles.py . All particles must be assigned a Z_2 parity and can have a flexible number of attributes, such as mass, spin, electric charge, etc. Two particles are considered equal if all their shared properties are equal.

Inclusive or *generic* particles are introduced by leaving one or more of their properties undefined. For instance, a Z_2-even particle with electric charge -1, spin 1/2 but undefined mass will be matched to electrons, muons and taus. This is useful when defining generic simplified models (*elements*) in the *Database*. All *inclusive* particles used by the *Database* are separately defined in databaseParticles.py . Examples are:

- 'l' for electrons, and muons,
- 'L' for electrons, muons, and taus,
- 'q' for u-, d-, and s-quarks,
- 'jet' for u-, d-, s-, c-quarks and gluons

[1] In order to treat the UL and EM map results on the same footing, SModelS applies a trivial binary efficiency to elements for UL-type results as will be explained in detail later.

- 'anyOdd' for any Z_2-odd particle
- '*' for any Z_2-even particle
- **Particles are described by the** Particle Class

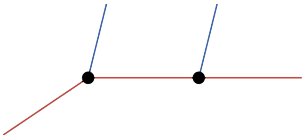
Vertices

Each Z_2-odd decay is represented by a vertex containing the outgoing states (one Z_2-odd state and the Z_2-even particles), as shown in the *scheme above*.

- **Vertices are described by the** ParticleList Class

Branches

A branch is the basic substructure of an *element*. It represents a series of cascade decays of a single initial Z_2-odd state. The diagram below illustrates an example of a branch.

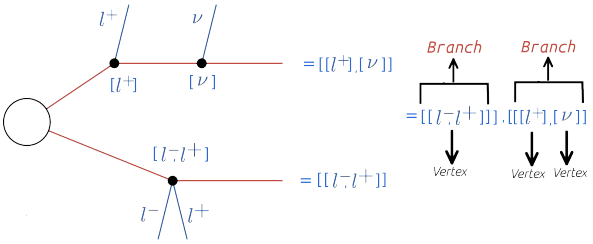


The structure of each branch is fully defined by its number of vertices and the number of *particles* coming out of each vertex.

- **Branches are described by the** Branch Class

Element Representation: Bracket Notation

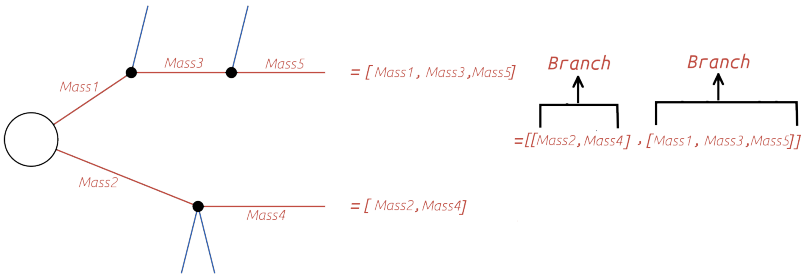
The structure and Z_2-even states of *elements* can be represented in a compact and textual form using a nested bracket notation. The scheme below shows how to convert between the graphical and bracket representations of an element:



The brackets are ordered and nested in the following way. The outermost brackets correspond to the *branches* of the *element*. The branches are sorted according to their size (see *element sorting*) and each branch contains an *ordered* list of *vertices*. Each vertex contains a list of the Z_2-even particles (represented by their label and sorted alphabetically) coming out of the vertex. Schematically, for the example in the *figure above*, we have:

```
element = [branch1, branch2]
   branch1 = [vertex1]
      vertex1 = [l+,l-]
   branch2 = [vertex1,vertex2]
      vertex1 = [l+]
      vertex2 = [nu]
```

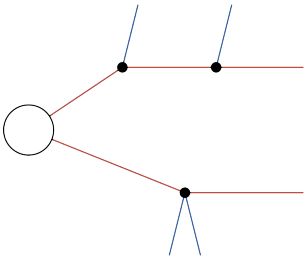
Although the above scheme can be useful and provides a simplified representation of an *element*, it provides no information about the Z_2-odd (BSM) states appearing in the *element*. However, information about a specific property of Z_2-odd states can also be represented in a nested bracket notation. For instance, all the masses of the BSM states in a given *element* can be represented as shown below:



Similar arrays can be built with any property (width, charge, spin, etc) of the Z_2-odd particles in an *element*.

Topologies

It is often useful to classify *elements* according to their overall structure or topology. Each topology corresponds to an *undressed element*, removed of its specific *particle* states. Therefore the topology is fully determined by its number of branches, number of vertices in each *branch* and number of *particles* coming out of each *vertex*. An example of a topology is shown below:



Within SModelS, elements are grouped according to their topology. Hence topologies represent a list of elements sharing a common basic structure (same number of branches, vertices and final states in each vertex).

- **Topologies are described by the** Topology Class

Database Definitions

The so-called experiment module contains the basic tools necessary for handling the database of experimental results. The SModelS database collects experimental results of searches from both ATLAS and CMS, which are used to compute the experimental constraints on specific models. Starting with version 1.1, the SModelS database includes two types of experimental constraints:

- Upper Limit (UL) constraints: constrains on $\sigma \times BR$ of simplified models, provided by the experimental collaborations (see *UL-type results*);
- Efficiency Map (EM) constraints: constrains the total signal ($\sum \sigma \times BR \times \epsilon$) in a specific signal region. Here ϵ denotes the acceptance times efficiency. These are either provided by the experimental collaborations or computed by theory groups (see *EM-type results*);

Although the two types of constraints above are very distinct, both the folder structure and the object structure of SModelS are sufficiently flexible to simutaneously handle both *UL-type* and *EM-type* results. Therefore, for both *UL-type* and *EM-type* constraints, the database obeys the following structure:

- *Database*: collects a list of *Experimental Results*.

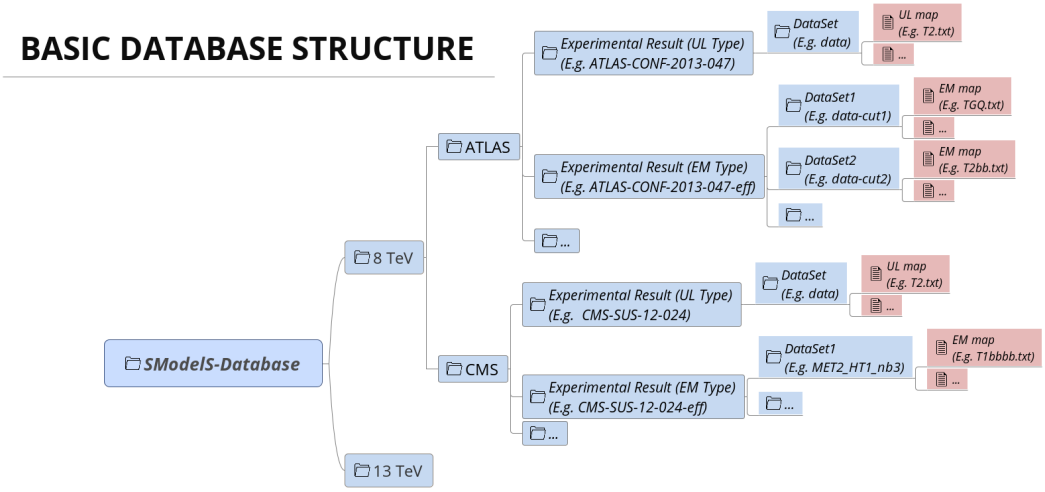
– *Experimental Result*: each *Experimental Result* corresponds to an experimental preliminary result (i.e. a CONF-NOTE or PAS) or publication and contains a list of *DataSets* as well as general information about the result (luminosity, publication reference,...).

  * *DataSet*: a single *DataSet* corresponds to one signal region of the experimental note or publication.[1] In case of *UL-type results* there is a single *DataSet*, usually corresponding to the best signal region or a combination of signal regions (for more details see *DataSet*). For *EM-type results*, there is one *DataSet* for each signal region. Each *DataSet* contains the Upper Limit maps for *Upper Limit results* *or* the Efficiency maps for *Efficiency Map results*.

    · Upper Limit map: contains the upper limit constraints for *UL-type results*. Each map contains upper limits for the signal cross-section for a single simplified model (or more precisely to a single *element* or sum of *elements*) as a function of the simplified model parameters.

    · Efficiency map: contains the efficiencies for *EM-type results*. Each map contains efficiencies for the signal for a single simplified model (or more precisely to a single *element* or sum of *elements*) as a function of the simplified model paramters.

A schematic summary of the above structure can be seen below:



In the following sections we describe the main concepts and elements which constitute the SModelS database. More details about the database folder structure and object struture can be found in *Database of Experimental Results*.

Database

Each publication or conference note can be included in the database as an *Experimental Result*. Hence, the database is simply a collection of experimental results.

- **The Database is described by the** Database Class

Experimental Result

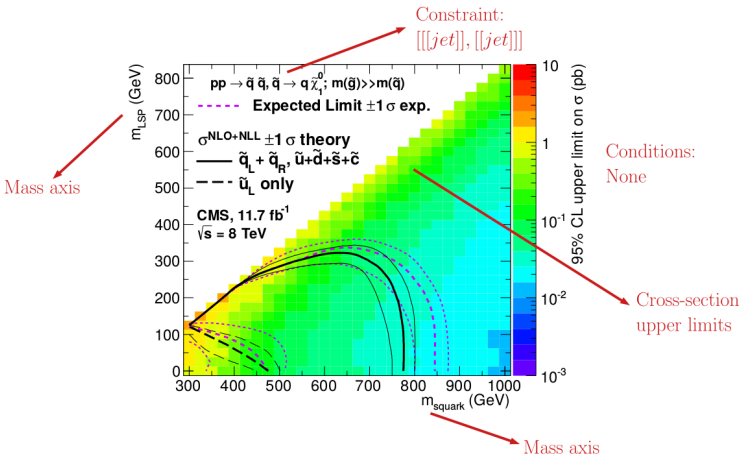
An experimental result contains all the relevant information corresponding to an experimental publication or preliminary result. In particular it holds general information about the experimental analysis, such as the corresponding luminosity, center of mass energy, publication reference, etc. The current version allows for two possible types of experimental results: one containing upper limit maps (*UL-type*) and one containing efficiency maps (*EM-type*).

[1] The name *Data Set* is used instead of signal region because its concept is slightly more general. For instance, in the case of *UL-type results*, a *DataSet* may not correspond to a single signal region, but to a combination of signal regions.

- **Experimental Results are described by the** ExpResult Class

Experimental Result: Upper Limit Type

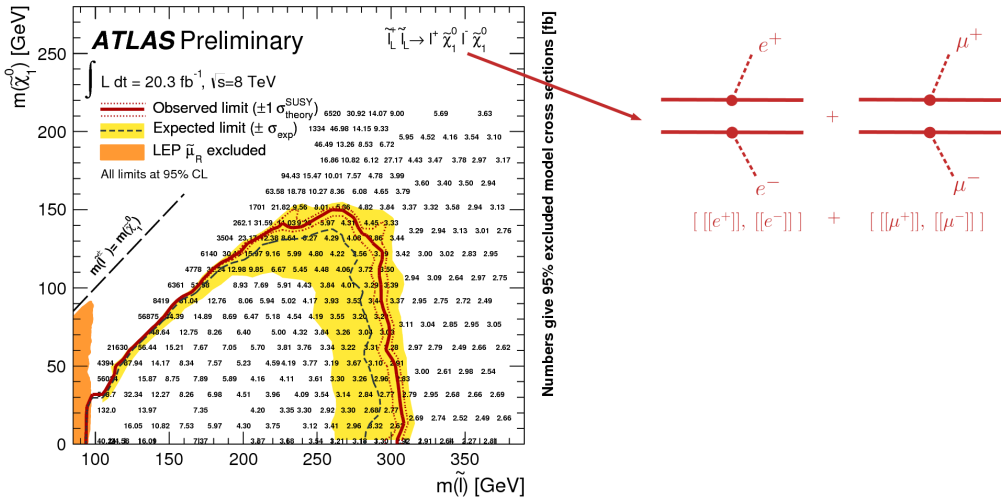
Upper Limit (UL) experimental results contain the experimental constraints on the cross section times branching ratio ($\sigma \times BR$) for Simplified Models from a specific experimental publication or preliminary result. These constraints are typically given in the format of Upper Limit maps, which correspond to 95% confidence level (C.L.) upper limit values on $\sigma \times BR$ as a function of the respective parameter space (usually BSM masses or slices over mass planes). The UL values typically refer to the best signal region (for a given point in parameter space) or a combination of signal regions. Hence, for UL results there is a single *DataSet*, containing one or more UL maps. An example of a UL map is shown below:



Within SModelS, the above UL map is used to constrain the simplified model $\tilde{q} + \tilde{q} \rightarrow \left(jet + \tilde{\chi}_1^0\right) + \left(jet + \tilde{\chi}_1^0\right)$. Using the SModelS notation this simplified model is mapped to the *element* $[[[jet]], [[jet]]]$, using the notation defined in *Bracket Notation*. The specific BSM states appearing in the simplified model are replaced by generic Z_2-even *particles* which have no attributes, except for its Z_2 parity. The only exception are the last BSM states appearing in the cascade decay, which signature can be specified through the final state property. If no final state is defined, the *element* is assumed to have a (MET, MET) final state signature. However, other signatures are also possible, such as *HSCP* (heavy stable charged particle), *R-hadrons*, etc. A list of all possible database BSM states (or *particles*) can be found in smodels/experiment/databaseParticles.py. Usually a single preliminary result/publication contains several UL maps, hence each UL-type experimental result contains several UL maps, each one constraining different simplified models (*elements* or sum of *elements*). *We also point out that the exclusion curve shown in the UL map above is never used by SModelS.*

Upper Limit Constraint

The upper limit constraint specifies which simplified model (represented by an *element* or sum of *elements*) is being constrained by the respective UL map. For simple constraints as the one shown in the *UL map* above, there is a single *element* being constrained ($[[[jet]], [[jet]]]$). In some cases, however, the constraint corresponds to a sum of *elements*. As an example, consider the ATLAS analysis shown below:



As we can see, the upper limits apply to the sum of the cross sections:

$$\sigma = \sigma([[[e^+]], [[e^-]]]) + \sigma([[[\mu^+]], [[\mu^-]]])$$

In this case the UL constraint is simply:

$$[[[e^+]], [[e^-]]] + [[[\mu^+]], [[\mu^-]]]$$

where it is understood that the sum runs over the weights of the respective *elements* and not over the *elements* themselves.

Note that the sum can be over particle charges, flavors or more complex combinations of elements. However, almost all experimental results sum only over elements sharing a common *topology*.

Finally, in some cases the UL constraint assumes specific contributions from each *element*. For instance, in the *example above* it is implicitly assumed that both the electron and muon *elements* contribute equally to the total cross section. Hence these conditions must also be specified along with the constraint, as described in *UL conditions*.

Upper Limit Conditions

When the analysis *constraints* are non-trivial (refer to a sum of elements), it is often the case that there are implicit (or explicit) assumptions about the contribution of each element. For instance, in the *figure above*, it is implicitly assumed that each lepton flavor contributes equally to the summed cross section:

$$\sigma([[[e^+]], [[e^-]]]) = \sigma([[[\mu^+]], [[\mu^-]]]) \quad \text{(condition)}$$

Therefore, when applying these constraints to general models, one must also verify if these conditions are satisfied. Once again we can express these conditions in *bracket notation*:

$$[[[e^+]], [[e^-]]] = [[[\mu^+]], [[\mu^-]]] \quad \text{(condition)}$$

where it is understood that the condition refers to the weights of the respective elements and not to the elements themselves.

In several cases it is desirable to relax the analysis conditions, so the analysis upper limits can be applied to a broader spectrum of models. Once again, for the example mentioned above, it might be reasonable to impose instead:

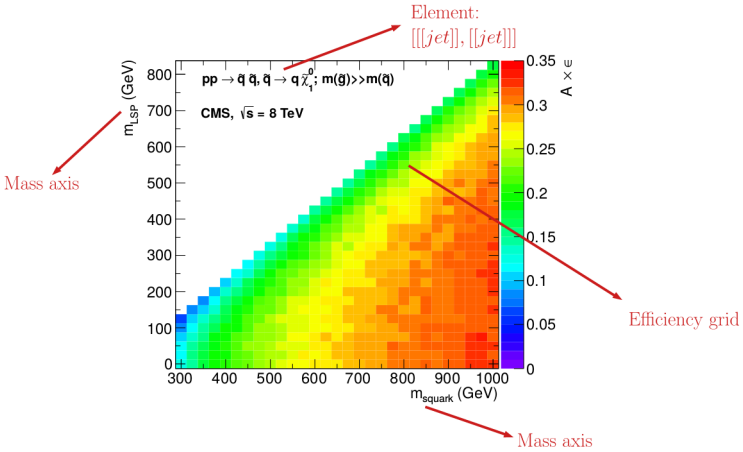
$$[[[e^+]], [[e^-]]] \simeq [[[\mu^+]], [[\mu^-]]] \quad \text{(fuzzy condition)}$$

The *departure* from the exact condition can then be properly quantified and one can decide whether the analysis upper limits are applicable or not to the model being considered. Concretely, SModelS computes for each condition a number between 0 and 1, where 0 means the condition is exactly satisfied and 1 means it is maximally violated. Allowing for a 20% violation of a condition corresponds approximately to a ''condition violation value" (or simply condition value) of 0.2. The condition values are given as an output of SModelS, so the user can decide what are the maximum acceptable values (see *maxcond* in the parameters file).

Experimental Result: Efficiency Map Type

Unlike *UL-type results*, the main information held by Efficiency Map (EM) results are the efficiencies for simplified models (represented by an *element* or sum of *elements*). These may be provided by the experimental collaborations or independently computed by theory groups. Efficiency maps correspond to a grid of simulated acceptance times efficiency ($A \times \epsilon$) values for a specific signal region. In the following we will refer to $A \times \epsilon$ simply as *efficiency* and denote it by ϵ. Furthermore, additional information, such as the luminosity, number of observed and expected events, etc is also stored in an EM-type result.

Another important difference between *UL-type results* and *EM-type results* is the existence of several signal regions, which in SModelS are mapped to *DataSets*. While *UL-type results* contain a single *DataSet* (''signal region"), EM results hold several *DataSets*, one for each signal region (see the *database scheme* above). Each *DataSet* contains one or more efficiency maps, one for each *element* or sum of *elements*. The efficiency map is usually a function of the BSM masses (or masses and widths) appearing in the element, as shown by the example below:



Within SModelS the above EM map is used to compute the efficiency for the *element* $[[[jet]], [[jet]]]$, where we are using the notation defined in *Bracket Notation*. As in the case of *UL-type results*, the specific BSM states appearing in the simplified model are replaced by generic Z_2-even *particles* which have no attributes, except for its Z_2 parity. The only exception are the last BSM states appearing in the cascade decay, which signature can be specified through the final state property. If no final state is defined, the *element* is assumed to have a (MET, MET) final state signature. However, other signatures are also possible, such as *HSCP* (heavy stable charged particle), *R-hadrons*, etc. A list of all possible database BSM states (or *particles*) can be found in smodels/experiment/databaseParticles.py. Usually there are several EM maps for a single *data set*: one for each *element* or sum of *elements*. In order to use a language similar to the one used in *UL-type results*, the *element* (or *elements*) for which the efficiencies correspond to are still called *constraint*.

Although efficiencis are most useful for *EM-type results*, their concept can also be extended to *UL-type results*. For the latter, the efficiencies for a given element are either 1, if the element appears in the *UL constraint*, or 0, otherwise. Atlhough trivial, this extension allows for a unified treatment of *EM-type results* and *UL-type results* (see *Theory Predictions* for more details).

Data Sets

Data sets are a way to conveniently group efficiency maps corresponding to the same signal region. As discussed in *UL-type results*, data sets are not necessary for UL-type results, since in this case there is a single ''signal region". Nonetheless, data sets are also present in *UL-type results* in order to allow for a similar structure for both *EM-type* and *UL-type* results (see *database scheme*).

For *UL-type results* the data set contains the UL maps as well as some basic information, such as the type of *Experimental Result* (UL). On the other hand, for *EM-type results*, each data set contains the EM maps for the corresponding signal region as well as some additional information: the observed and expected number of events in the signal region, the signal upper limit, etc. In the folder structure shown in *database scheme*, the upper limit maps and efficiency maps for each *element* (or sum of *elements*) are stored in files labeled accoring to the *TxName convention*.

- **Data Sets are described by the** DataSet Class

TxName Convention

Since using the *bracket notation* to describe the simplified models appearing in the upper limit or efficiency maps can be rather lenghty, it is useful to define a shorthand notation for the *constraints*. SModelS adopts a notation based on the CMS SMS conventions, where each specific *constraint* is labeled as *T<constraint name>*, which we refer as *TxName*. For instance, the TxName corresponding to the constraint in the *example above* is *TSlepSlep*. A complete list of TxNames can be found here.

- **Upper limit and efficiency maps are described by the** TxName Class

More details about the database folder structure and object structure can be found in Database of Experimental Results.

SModelS Structure

The main ingredients relevant for SModelS are:

Basic Input

Basic Model Input

SModelS requires two types of input from the user:

- the *particle content* of the BSM model (BSM states and their quantum numbers) and
- the *model parameters*, such as masses, widths, branching ratios and cross-sections.

Below we describe how this information should be provided.

- **Information about the SM and BSM particles, along with their cross-sections is stored in a** Model object

Particle Content

The definition of the BSM states and their quantum numbers can be provided in either of the following two formats:

- as a python module, as illustrated in `mssm.py`,
- or as an SLHA file containing *QNUMBERS blocks* (see `mssmQNumbers.slha`).

The python module is more flexible and allows the user to define their own particle properties. This format is automatically generated by the micrOMEGAs interface to SModelS. The SLHA file can be automatically generated by tools relying on the UFO format, in particular MadGraph . A path to the user's own model file (in either format) can be specified in the *parameter file*, in the *[particles]* section.

Model Parameters

Once the *particle content* has been specified, the main input for SModelS is the set of model parameters (masses, widths,...), which can be given in the two following forms:

- an SLHA (SUSY Les Houches Accord) file containing masses, widths, branching ratios and cross sections for the BSM states (see an example file `here`)
- an LHE (Les Houches Event) file containing parton level events (see an example file `here`)

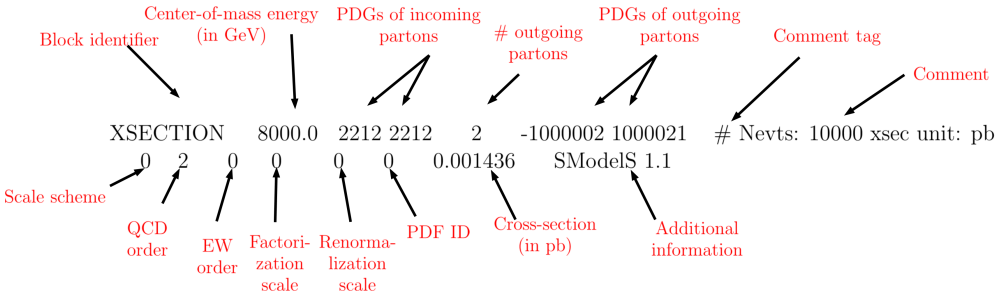
The SLHA format is usually more compact and best suited for supersymmetric models. On the other hand, an LHE file can always be generated for any BSM model (through the use of your favorite MC generator).[1] In this case, however, the precision of the results is limited to the MC statistics used to generate the file.

In the case of SLHA input only, the production cross sections for the BSM states also have to be included in the SLHA file as SLHA blocks, according to the *SLHA cross section format* (see `example file`). For the MSSM and some of its extensions, they may be calculated automatically using Pythia and NLLfast, as discussed in *cross section calculator*.

In the case of LHE input, the total production cross section as well as the center-of-mass energy should be listed in the <init></init> block, according to the standard LHE format (see `example file`). Moreover, all the Z_2-even *particles* should be set as stable, since in SModelS they are effectively considered as final states. When generating the events it is also important to ensure that no mass smearing is applied, so the mass values for a given particle are the same throughout the LHE file. We also point out that all the decays appearing in the LHE input are assumed to be prompt, so this input format is not well suited if the model contains meta-stable particles. An example of how to add the width information after reading an LHE input file can be found in this notebook .

SLHA Format for Cross Sections

A list of cross section blocks (one for each production process) must be included in the SLHA file for the SLHA-based decomposition. The SLHA format for each cross section block is shown below (see the Les Houches note):



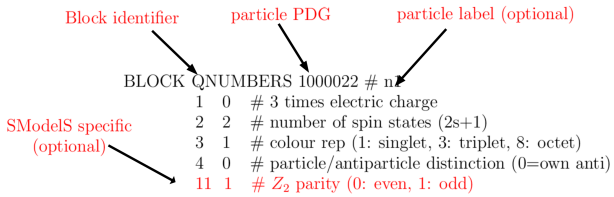
The above example shows the cross section for $pp \to \tilde{u}_L^* + \tilde{g}$ at a center-of-mass energy of 8 TeV and at leading order. The only information used by SModelS are the center-of-mass energy, the outgoing particle PDGs, the cross section value and the QCD order. *If the input file contains two cross sections for the same process but at different QCD orders, only the highest order will be used.*

- **Reading of cross sections from an input file is implemented by the** getXsecFromSLHAFile **method**

[1] SModelS can easily be used for non-SUSY models as long as they present a Z_2-type symmetry. However, it is the responsibility of the user to make sure that the SMS results in the database actually apply to the model under consideration.

SLHA Format for Quantum Numbers

If the *particle content* of the input model is specificed through an SLHA file (instead of a python module), it must contain a QNUMBERS block for each particle following the format below:



This specifies the particle PDG, electric charge, color representation and spin. Furthermore, if the first line contains a comment (after the particle PDG), it will be used as the particle label, otherwise the particle label will be its PDG number. The entry number 5 (Z_2 parity) is a SModelS specific line which can be added to specify the *particle* parity (even or odd). If this line is missing the parity will be assumed to be odd. Finally, if the particle is not its own anti-particle (specified by entry number 4), a second particle will be added to the model with the opposite electric charge and minus the PDG number.

LHE-reader

More general models can be input through an LHE event file containing parton-level events, including the production of the primary mothers and their cascade decays. The LHE-reader goes through the events and by doing so creates dictionaries mapping the different particles to their masses and decays which corresponds to the DECAY and MASS blocks of the SLHA file. The pair production cross sections are obtained by adding up the weights of all events with a the same pair of mother particles (see crossSection.getXsecFromLHEFile).

Notice that, for the LHE decomposition, the *elements* generated are restricted to the events in the input file. Hence, the uncertainties on the elements weights (and which *elements* are actually generated by the model) are fully dependent on the Monte Carlo statistics used to generate the LHE file. Also, when generating the events it is important to ensure that no mass smearing is applied, so the events always contain the same mass value for a given particle.

Note that since all decays appearing in an LHE event are assumed to be prompt, the LHE-based decomposition is not well suited if the model contains meta-stable BSM particles. If needed, the user can manually add the width information as shown in this notebook example .

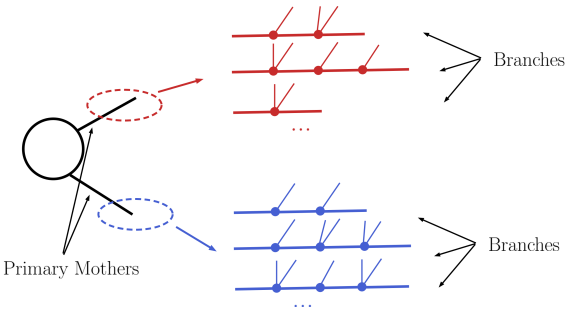
- **The LHE reader is implemented by the** LHE reader method

Decomposition into Simplified Models

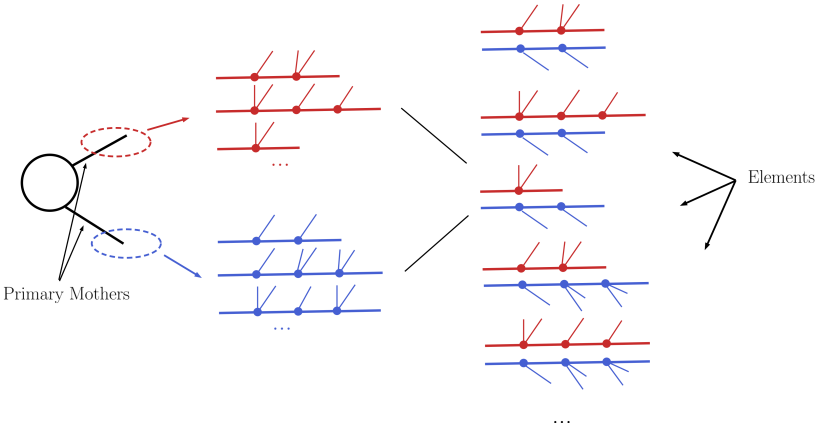
Given an input model (see *Basic Input*), the first task of SModelS is to decompose the full model into a sum of Simplified Models (or *elements* in SModelS language).

Decomposition

The input model stores information about the masses of all the BSM states as well as their production cross sections, decay branching ratios and total widths. All the cross sections for *production of two* Z_2-odd *states* serve as the initial step for the decomposition. (All the other cross sections with a different number of Z_2-odd states are ignored.) Starting from these primary mothers, all the possible decays are generated according to the decay information for each particle. This procedure is represented in the figure below:



Each of the possible cascade decays for each mother corresponds to a *branch*. In order to finally generate *elements*, all the branches are combined in pairs according to the production cross sections, as shown below:



For instance, assume [b1,b2,b3] and [B1,B2] represent all possible branches (or cascade decays) for the primary mothers A and B, respectively. Then, if a production cross section for $pp \to A + B$ is given in the input file, the following elements will be generated:

[b1,B1], [b1,B2], [b2,B1], [b2,B2], [b3,B1] and [b3,B2]

The decomposition process either stops when the lightest neutral BSM particle or a stable (or meta-stable) particle is reached. Meta-stable particles are the ones with a total decay width smaller than a certain cutoff value (set by *promptWidth in the parameters file*). For these states, in addition to the *branches* where this particle decays, the decomposition will generate *branches* where this particle appears as a final state (undecayed).

Each of the *elements* generated according to the procedure just described will also store its weight, which equals its production cross section times all the branching ratios appearing in it. In order to avoid a too large number of elements, only those satisfying a *minimum weight* requirement are kept. Furthermore, the elements are grouped according to their *topologies*. The final output of the decomposition is a list of such topologies, where each topology contains a list of the elements generated during the decomposition.

- **The decomposition is implemented by the** decompose method

Minimum Decomposition Weight

Some models may contain a large number of new states and each may have a large number of possible decays. As a result, long cascade decays are possible and the number of elements generated by the decomposition process may become too large, and the computing time too long. For most practical purposes, however, elements with extremely small weights (cross section times BRs) can be discarded, since they will fall well below the experimental limits. Therefore, during the decomposition, whenever an element is generated with a weight below some minimum value, this element (and all elements derived from it) is ignored. The minimum weight to be considered is set by the *sigmacut parameter in the parameters file* and is easily adjustable (see decomposer.decompose).

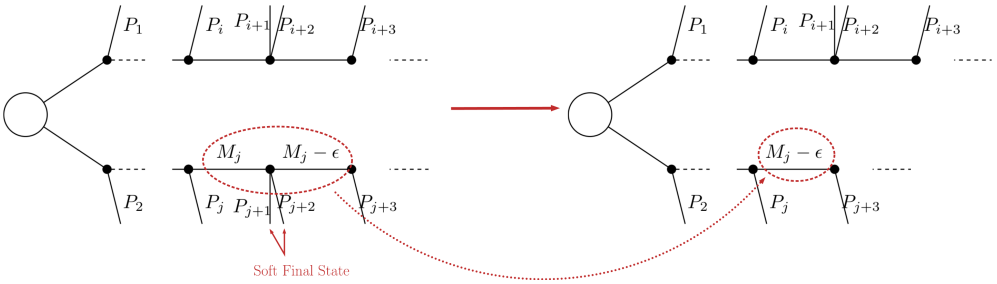
Note that, when computing the *theory predictions*, the weight of several *elements* can be combined together. Hence it is recommended to set the value of *sigmacut* approximately one order of magnitude below the minimum signal cross sections the experimental data can constrain.

Compression of Elements

During the decomposition process it is possible to perform several simplifications on the *elements* generated. Two useful simplifications are possible: *Mass Compression* and *Invisible Compression*. The main advantage of performing these compressions is that the simplified *element* is always shorter (has fewer cascade decay steps), which makes it more likely to be constrained by experimental results. The details behind the compression methods are as follows:

Mass Compression

In case of small mass differences, the *prompt* decay[1] of a BSM *particle* to a nearly degenerate one will in most cases produce soft final states, which can not be experimentally detected. Consequently, it is a good approximation to neglect the soft final states and *compress* the respective decay, as shown below:



After the compression, only the lightest of the two near-degenerate masses are kept in the element, as shown *above*. The main parameter which controls the compression is *minmassgap*, which corresponds to the maximum value of ϵ in the *figure above* to which the compression is performed:

if $|M_j - M_{j+1}| < minmassgap \rightarrow$ the *prompt* decay is compressed
if $|M_j - M_{j+1}| > minmassgap$ or the decay is not prompt $\rightarrow$ the decay is NOT compressed

Note that the compression is an approximation since the final states, depending on the boost of the parent state, may not always be soft. It is recommended to choose values of *minmassgap* between 1-10 GeV; the default value is 5 GeV.

- **Mass compression is implemented by the** massCompress **method** and can be easily turned on/off by the *doCompress parameter in the parameters file*.

Invisible Compression

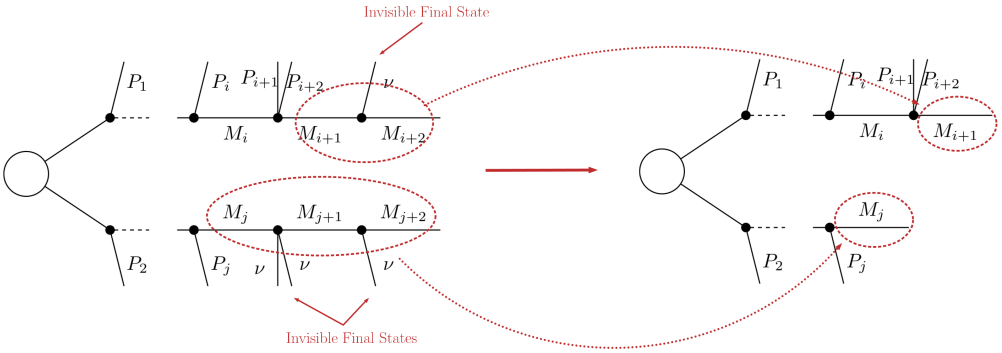
Another type of compression is possible when the last BSM decay appearing in a *branch* is invisible. The most common example is

$$A \rightarrow \nu + B$$

as the last step of the decay chain, where ν is a SM neutrino, A is a *neutral particle and/or decays promptly* and B is an invisible particle leading to a MET signature. Since both the neutrino and B are invisible, for all experimental purposes the effective MET object is $B + \nu = A$. Hence it is possible to omit the last step in the cascade decay,

[1] Decays of meta-stable particles should not be compressed, even if soft, since they might result in distinct signatures depending on the quantum numbers of the decaying particle. Particles are assumed to be meta-stable if their width is below the value set by the *promptWidth parameter*.

resulting in a compressed element. Note that this compression can be applied consecutively to several steps of the cascade decay if all of them contain only invisible final states:



After the compression, the last BSM state appearing in the compressed *element* is replaced by an effective *particle* with no electric or color charge, with label "inv" and with the mass of the parent (A in the example above). Furthermore since the original neutral final state (B) can in principle be meta-stable, the new effective final state inherits its width.

- **Invisible compression is implemented by the** invisibleCompress **method** and can be easily turned on/off by the *doInvisible parameter in the parameters file*.

Element Sorting

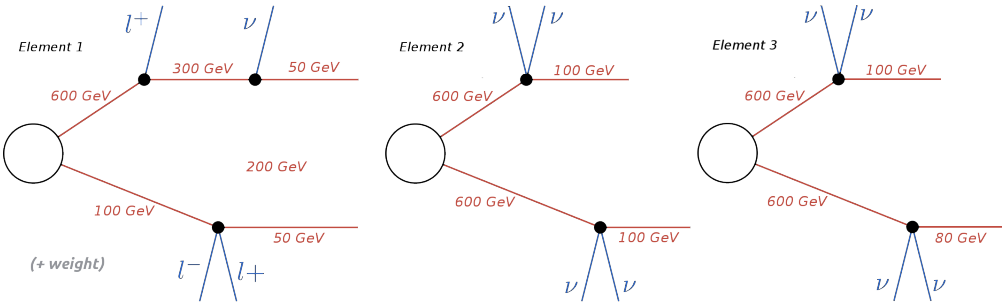
In order to improve the code performance, *elements* created during *decomposition* and sharing a common *topology* are sorted. Sorting allows for an convenient ordering of the elements belonging to a topology and faster element comparison. Elements are sorted according to their branches. Branches are compared according to the following order of properties:

- Number of vertices
- Number of final states in each vertex
- BSM (Z_2-odd) particles
- Z_2-even final state particles in each vertex

Finally, particles are compared according to the following order of properties (if defined):

- Z_2 parity
- Spin
- Color representation
- Electric charge
- Mass
- Total width

As an example, consider the three elements below where all BSM (Z_2-odd) particles only differ by their mass:



The correct ordering of the above elements is:

Element 3 < Element 2 < Element 1

Element 1 is 'larger' than the other two since it has a larger number of vertices. Elements 2 and 3 are identical, except for their masses. Since the last BSM particle appearing in the lower branch of Element 3 has a smaller mass than the corresponding particle in Element 2, the former is 'smaller' than the latter. Finally if all the branch features listed above are identical for both branches, the elements being compared are considered to be equal. Futhermore, the branches belonging to the same element are also sorted. Hence, if an element has two branches:

$$element = [branch1, branch2],$$

it implies

$$branch1 < branch2$$

regarding their ordering.

- **Branch sorting is implemented by the** sortBranches **method**

Theory Predictions

The *decomposition* of the input model as a sum of *elements* (simplified models) is the first step for confronting the model with the experimental limits. The next step consists of computing the relevant signal cross sections (or *theory predictions*) for comparison with the experimental limits. Below we describe the procedure for the computation of the theory predictions after the model has been decomposed.

Computing Theory Predictions

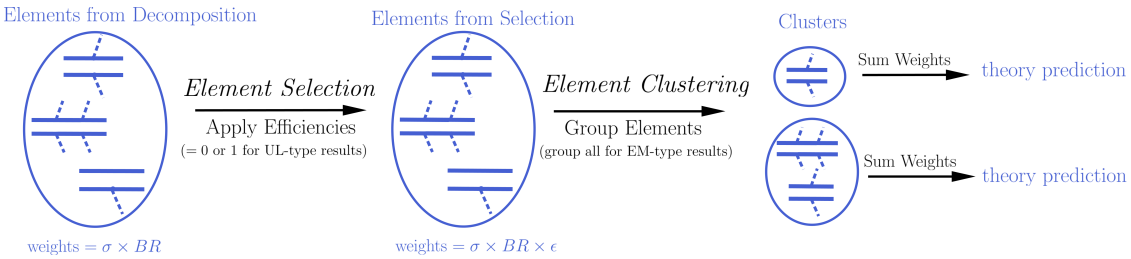
As discussed in *Database Definitions*, the SModelS database allows for two types of experimental constraints: Upper Limit constraints (see *UL-type results*) and Efficiency Map constraints (see *EM-type results*). Each of them requires different theoretical predictions to be compared against experimental data.

UL-type results constrains the weight ($\sigma \times BR$) of one *element* or sum of *elements*. Therefore SModelS must compute the theoretical value of $\sigma \times BR$ summing only over the *elements* appearing in the respective *constraint*. This is done by assigning an efficiency equal to 1 (0) to each element, if the element appears (does not appear) in the *constraint*. Then the final theoretical prediction is the sum over all *elements* with a non-zero value of $\sum \sigma \times BR \times \epsilon$. This value can then be compared with the respective 95% C.L. upper limit extracted from the UL map (see *UL-type results*).

On the other hand, *EM-type results* constrain the total signal ($\sum \sigma \times BR \times \epsilon$) in a given signal region (*DataSet*). Consequently, in this case SModelS must compute $\sigma \times BR \times \epsilon$ for each *element*, using the efficiency maps for the corresponding *DataSet*. The final theoretical prediction is the sum over all *elements* with a non-zero value of $\sigma \times BR \times \epsilon$. This value can then be compared with the signal upper limit for the respective signal region (*data set*).

For experimental results for which the covariance matrix is provided, it is possible to combine all the signal regions (see *Combination of Signal Regions*). In this case the final theory prediction corresponds to the sum of $\sigma \times BR \times \epsilon$ over all signal regions (and all elements) and the upper limit is computed for this sum.

Although the details of the theoretical prediction computation differ depending on the type of *Experimental Result* (*UL-type results* or *EM-type results*), the overall procedure is common for both types of results. Below we schematically show the main steps of the theory prediction calculation:



As shown above the procedure can always be divided in two main steps: *Element Selection* and *Element Clustering*. Once the *elements* have been selected and clustered, the theory prediction for each *DataSet* is given by the sum of all the *element* weights ($\sigma \times BR \times \epsilon$) belonging to the same cluster:

$$\text{theory prediction} = \sum_{cluster} (\text{element weight}) = \sum_{cluster} (\sigma \times BR \times \epsilon)$$

Below we describe in detail the *element selection* and *element clustering* methods for computing the theory predictions for each type of *Experimental Result* separately.

- **Theory predictions are computed using the** theoryPredictionsFor **method**

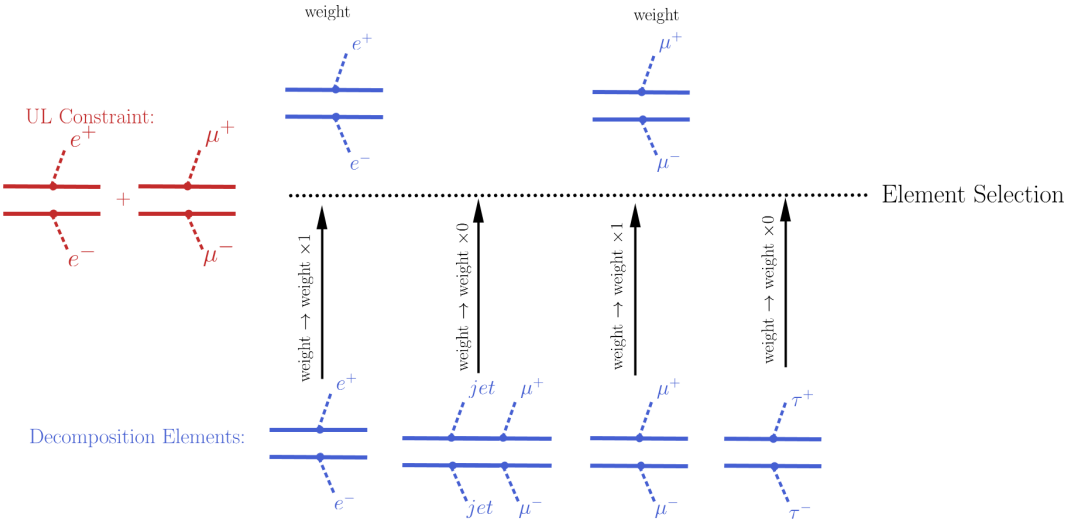
Theory Predictions for Upper Limit Results

Computation of the signal cross sections for a given *UL-type result* takes place in two steps. First selection of the *elements* generated by the model *decomposition* and then clustering of the selected elements according to their properties. These two steps are described below.

Element Selection

An *UL-type result* holds upper limits for the cross sections of an *element* or sum of *elements*. Consequently, the first step for computing the theory predictions for the corresponding experimental result is to select the *elements* that appear in the *UL result constraint*. This is conveniently done attributing to each *element* an efficiency equal to 1 (0) if the *element* appears (does not appear) in the *constraint*. After all the *elements* weights ($\sigma \times BR$) have been rescaled by these ''trivial" efficiencies, only the ones with non-zero weights are relevant for the signal cross section. The *element* selection is then trivially achieved by selecting all the *elements* with non-zero weights.

The procedure described above is illustrated graphically in the figure below for the simple example where the *constraint* is $[[[e^+]], [[e^-]]] + [[[\mu^+]], [[\mu^-]]]$.

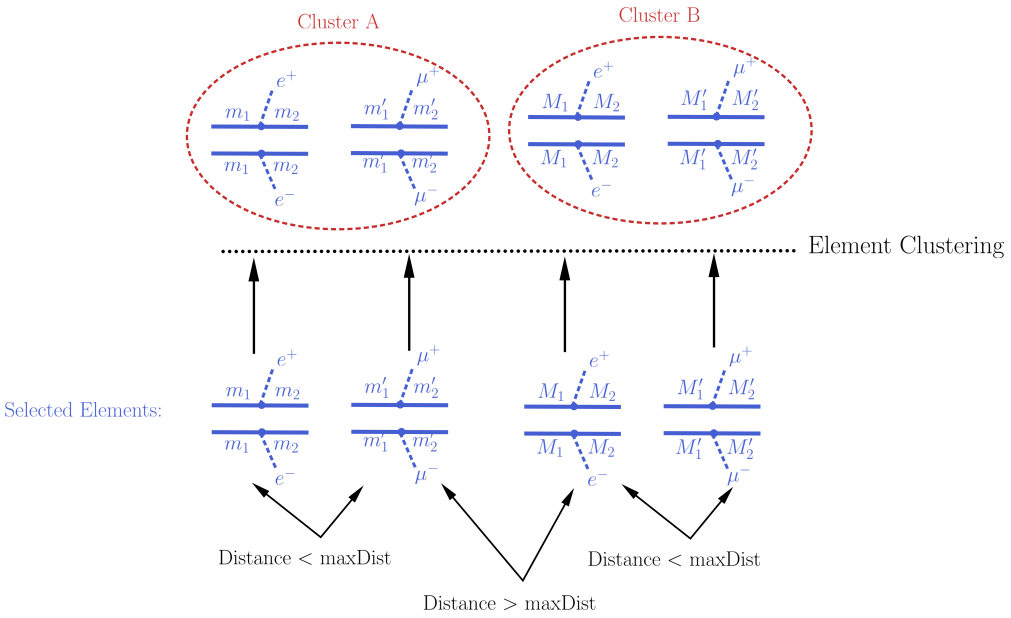


- **The element selection is implemented by the** getElementsFrom **method**

Element Clustering

Naively one would expect that after all the *elements* appearing in the *constraint* have been selected, it is trivial to compute the theory prediction: one must simply sum up the weights ($\sigma \times BR$) of all the selected *elements*. However, the selected *elements* usually differ in their masses and/or widths[1] and the experimental limit (see *Upper Limit constraint*) assumes that all the *elements* appearing in the *constraint* have the same efficiency, which typically implies that the distinct elements have the same mass arrays and widths. As a result, the selected *elements* must be grouped into *clusters* of equal masses and widths. When grouping the *elements*, however, one must allow for small differences, since the experimental efficiencies should not be strongly sensitive to tiny changes in mass or width values. For instance, assume two *elements* contain identical mass arrays, except for the parent masses which differ by 1 MeV. In this case it is obvious that for all experimental purposes the two *elements* have the same mass and should contribute to the same theory prediction (e.g. their weights should be added when computing the signal cross section). Unfortunately there is no way to unambiguously define ''similar efficiencies" or ''similar masses and widths" and the definition should depend on the *Experimental Result*. In the simplest case where the upper limit result corresponds to a single signal region (which is not always the case), one could assume that each element efficiency is inversely proportional to its upper limit. Hence the distance between two *elements* can be defined as the relative difference between their upper limits, as described in *element distance*. Then, if the *distance* between two selected *elements* is smaller than a maximum value (defined by maxDist), they are gouped in the same cluster and their cross-sections will be combined, as illustrated by the example below:

[1] When refering to an *element* mass or width, we mean all the masses and widths of the Z_2-odd *particles* appearing in the *element*. Two *elements* are considered to have identical masses and widths if their mass arrays and width arrays are identical.



Notice that the above definition of distance quantifies the experimental analysis sensitivity to changes in the *element* properties (masses and widths), which should correspond to changes in the upper limit value for the *element*. However, most *Experimental Results* combine distinct signal regions or use a more complex analysis in order to derive upper limits. In such cases, two *elements* can have (by chance) the same upper limit value, but still have very distinct efficiencies and should not be considered similar and combined. In order to deal with such cases an additional requirement is imposed when clustering two *elements*: the distance between both elements to the *average element* must also be smaller than maxDist . The *average element* of a list of *elements* corresponds to the *element* with the same common topology and final states, but with the mass array and widths replaced by the average mass and width over all the elements in the list. If this *average element* has an upper limit similar to all the *elements* in the list, we assume that all the elements have similar efficiencies and can be considered as similar to the given *Experimental Result*.

Once all the *elements* have been clustered, their weights can finally be added together and compared against the experimental upper limit.

- **The clustering of elements is implemented by the** clusterElements **method**.

Distance Between Elements

As mentioned *above*, in order to cluster the *elements* it is necessary to determine whether two *elements* are similar for a given *Experimental Result*. This usually means that both *elements* have similar efficiencies for the *Experimental Result*. Since an absolute definition of ''similar elements" is not possible and the sensitivity to changes in the mass or width of a given *element* depends on the experimental result, SModelS uses an ''upper limit map-dependent" definition. Each *element* is mapped to its corresponding upper limit for a given *Experimental Result* and the distance between two elements is simply given by the relative distance between the upper limits:

$$\text{Upper Limit(Element } A) = x, \quad \text{Upper Limit(Element } B) = y$$
$$\Rightarrow \text{distance}(A, B) = \frac{|x - y|}{(x + y)/2}$$

Theory Predictions for Efficiency Map Results

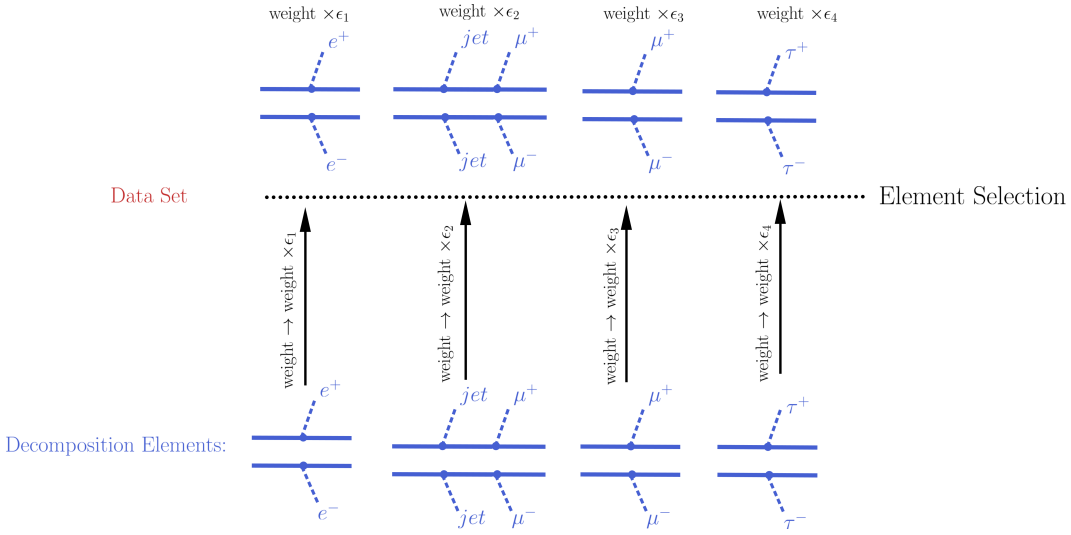
In order to compute the signal cross sections for a given *EM-type result*, so it can be compared to the signal region limits, it is first necessary to apply the efficiencies (see *EM-type result*) to all the *elements* generated by the model

decomposition. Notice that typically a single *EM-type result* contains several signal regions (*DataSets*) and there will be a set of efficiencies (or efficiency maps) for each *data set*. As a result, several theory predictions (one for each *data set*) will be computed. This procedure is similar (in nature) to the *Element Selection* applied in the case of a *UL-type result*, except that now it must be repeated for several *data sets* (signal regions).

After the *element*'s weights have being rescaled by the corresponding efficiencies for the given *data set* (signal region), all of them can be grouped together in a single cluster, which will provide a single theory prediction (signal cross section) for each *DataSet*. Hence the *element clustering* discussed below is completely trivial. On the other hand the *element selection* is slightly more involved than in the *UL-type result* case and will be discussed in more detail.

Element Selection

The element selection for the case of an *EM-type result* consists of rescaling all the *elements* weights by their efficiencies, according to the efficiency map of the corresponding *DataSet*. The efficiency for a given *DataSet* depends both on the *element* topology and its *particle* content. In practice the efficiencies for most of the *elements* will be extremely small (or zero), hence only a subset effectively contributes after the element selection[2]. In the figure below we illustrate the element selection for the case of an *EM-type result*/*DataSet*:



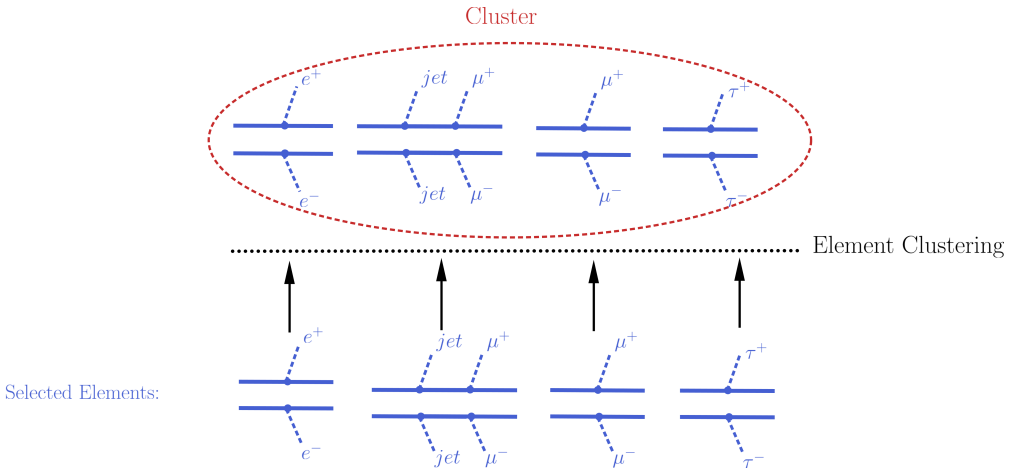
If, for instance, the analysis being considered vetoes $jets$ and τ's in the final state, we will have ϵ_2, $\epsilon_4 \simeq 0$ for the example in the *figure above*. Also, if the experimental result applies only to prompt decays and the *element* contains intermediate meta-stable BSM particles, its efficiency will be very small (although not necessarily zero).

- **The element selection is implemented by the** getElementsFrom **method**

Element Clustering

After the efficiencies have been applied to the element's weights, all the *elements* can be combined together when computing the theory prediction for the given *DataSet* (signal region). Since a given signal region correspond to the same signal upper limit for any *element*, the *distance* between any two *elements* for an *EM-type result* is always zero and the clustering procedure described *above* will trivially group together all the selected *elements* into a single cluster:

[2] The number of *elements* passing the selection also depends on the availability of efficiency maps for the *elements* generated by the decomposition. Whenever there are no efficiencies available for an element, the efficiency is taken to be zero.



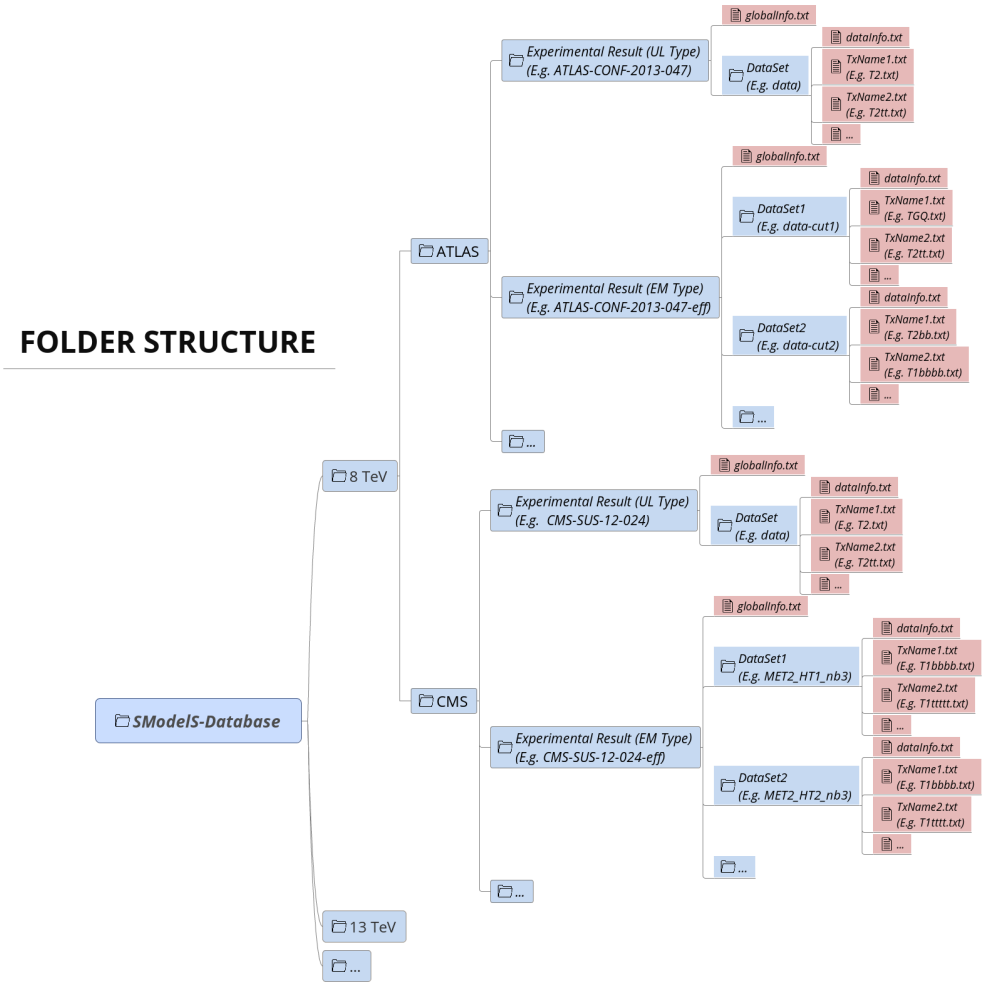
- **The clustering of elements is implemented by the** clusterElements **method**.

Database of Experimental Results

SModelS stores all the information about the experimental results in the *Database*. Below we describe both the *directory*, *object* structure of the *Database* and *how the information in stored in the database is used within SModelS*.

Database: Directory Structure

The *Database* is organized as files in an ordinary (UNIX) directory hierarchy, with a thin Python layer serving as the access to the database. The overall structure of the directory hierarchy and its contents is depicted in the scheme below (click to enlarge):



As seen above, the top level of the SModelS database categorizes the analyses by LHC center-of-mass energies, $\sqrt{s}$:

- 8 TeV
- 13 TeV

Also, the top level directory contains a file called `version` with the version string of the database. The second level splits the results up between the different experiments:

- 8TeV/CMS/
- 8TeV/ATLAS/

The third level of the directory hierarchy encodes the *Experimental Results*:

- 8TeV/CMS/CMS-SUS-12-024
- 8TeV/ATLAS/ATLAS-CONF-2013-047
- ...
- **The Database folder is described by the** Database Class

Experimental Result Folder

Each *Experimental Result* folder contains:

- a folder for each *DataSet* (e.g. `data`)
- a `globalInfo.txt` file

The `globalInfo.txt` file contains the meta information about the *Experimental Result*. It defines the center-of-mass energy $\sqrt{s}$, the integrated luminosity, the id used to identify the result and additional information about the source of the data. Here is the content of CMS-SUS-12-024/globalInfo.txt as an example:

```
sqrts: 8.0*TeV
lumi: 19.4/fb
id: CMS-SUS-12-024
prettyName: \slash{E}_{T}+b
url: https://twiki.cern.ch/twiki/bin/view/CMSPublic/PhysicsResultsSUS12024
arxiv: http://arxiv.org/abs/1305.2390
publication: http://www.sciencedirect.com/science/article/pii/S0370269313005339
contact: Keith Ulmer <keith.ulmer@cern.ch>, Josh Thompson <joshua.thompson@cern.ch>,␣
↪Alessandro Gaz <alessandro.gaz@cern.ch>
private: False
implementedBy: Wolfgang Waltenberger
lastUpdate: 2015/5/11
```

- **Experimental Result folder is described by the** ExpResult Class
- **globalInfo files are descrived by the** Info Class

Data Set Folder

Each *DataSet* folder (e.g. `data`) contains:

- the Upper Limit maps for *UL-type results* or Efficiency maps for *EM-type results* (`TxName.txt` files)
- a `dataInfo.txt` file containing meta information about the *DataSet*
- **Data Set folders are described by the** DataSet Class
- **TxName files are described by the** TxName Class
- **dataInfo files are described by the** Info Class

Data Set Folder: Upper Limit Type

Since *UL-type results* have a single dataset (see *DataSets*), the info file only holds some trivial information, such as the type of *Experimental Result* (UL) and the dataset id (None for UL-type results). Here is the content of CMS-SUS-12-024/data/dataInfo.txt as an example:

```
dataType: upperLimit
dataId: None
```

Data Set Folder: Efficiency Map Type

For *EM-type results* the `dataInfo.txt` contains relevant information, such as an id to identify the *DataSet* (signal region), the number of observed and expected background events for the corresponding signal region and the respective signal upper limits. Here is the content of CMS-SUS-13-012-eff/3NJet6_1000HT1250_200MHT300/dataInfo.txt as an example:

```
dataType: efficiencyMap
dataId: 3NJet6_1000HT1250_200MHT300
observedN: 335
expectedBG: 305
bgError: 41
upperLimit: 5.681*fb
expectedUpperLimit: 4.585*fb
```

TxName Files

Each *DataSet* contains one or more `TxName.txt` file storing the bulk of the experimental result data. For *UL-type results*, the TxName file contains the UL maps for a given simplified model (*element* or sum of *elements*), while for *EM-type results* the file contains the simplified model efficiencies. In addition, the TxName files also store some meta information, such as the source of the data and the *type* of result (*prompt* or *displaced*). If not specified, the type will be assumed to be prompt.[1] For instance, the first few lines of CMS-SUS-12-024/data/T1tttt.txt read:

```
txName: T1tttt
constraint: [[['t','t']],[['t','t']]]
condition: None
conditionDescription: None
figureUrl: https://twiki.cern.ch/twiki/pub/CMSPublic/PhysicsResultsSUS12024/T1tttt_
↪exclusions_corrected.pdf
source: CMS
validated: True
axes: [[x, y], [x, y]]
```

As seen above, the first block of data in the file contains information about the *element* or simplified model ([[[‘t’,’t’]],[[‘t’,’t’]]]) in *bracket notation* for which the data refers to as well as reference to the original data source and some additional information. The simplified model is assumed to contain neutral BSM final states (MET signature) and arbitrary (Z_2-odd *particles*) intermediate BSM states. If the experimental result refers to non-MET final states, the finalState field must list the type of BSM *particles* (see *UL-type* for more details). An example from the CMS-EXO-12-026/data/THSCPM1b.txt file is shown below:

```
txName: THSCPM1b
constraint: [[],[]]
source: CMS
axes: [[x], [x]]
finalState: ['HSCP', 'HSCP']
```

In addition, if specific BSM intermediate states are required, the intermediateState field must include a nested list (one for each branch) specifying the labels of the intermediate BSM states.[3] If the intermediateState field is specified, the corresponding result will only be applied to simplified models containing intermediate BSM particles with the same quantum numbers. One example is shown below:

```
txName: TExample
constraint: [[['q','q'],['W+']],[['q','q'],['W-']]]
finalState: ['MET', 'MET']
intermediateState: [['gluino','C1+'], ['gluino','C1-']]
```

The second block of data in the `TxName.txt` file contains the upper limits or efficiencies as a function of the relevant simplified model parameters:

[1] Prompt results are all those which assumes all decays to be prompt and the last BSM particle to be stable (or decay outside the detector). Searches for heavy stable charged particles (HSCPs), for instance, are classified as *prompt*, since the HSCP is assumed to decay outside the detector. Displaced results on the other hand require at least one decay to take place inside the detector.

[3] Although the finalState and intermediateState fields could be combined into a single entry, they are kept separate for backward compatibility.

```
upperLimits: [[[[4.0000E+02*GeV,0.0000E+00*GeV],[4.0000E+02*GeV,0.0000E+00*GeV]],1.
↪8158E+00*pb],
[[[4.0000E+02*GeV,2.5000E+01*GeV],[4.0000E+02*GeV,2.5000E+01*GeV]],1.8065E+00*pb],
[[[4.0000E+02*GeV,5.0000E+01*GeV],[4.0000E+02*GeV,5.0000E+01*GeV]],2.1393E+00*pb],
[[[4.0000E+02*GeV,7.5000E+01*GeV],[4.0000E+02*GeV,7.5000E+01*GeV]],2.4721E+00*pb],
[[[4.0000E+02*GeV,1.0000E+02*GeV],[4.0000E+02*GeV,1.0000E+02*GeV]],2.9297E+00*pb],
[[[4.0000E+02*GeV,1.2500E+02*GeV],[4.0000E+02*GeV,1.2500E+02*GeV]],3.3874E+00*pb],
[[[4.0000E+02*GeV,1.5000E+02*GeV],[4.0000E+02*GeV,1.5000E+02*GeV]],3.4746E+00*pb],
[[[4.0000E+02*GeV,1.7500E+02*GeV],[4.0000E+02*GeV,1.7500E+02*GeV]],3.5618E+00*pb],
[[[4.2500E+02*GeV,0.0000E+00*GeV],[4.2500E+02*GeV,0.0000E+00*GeV]],1.3188E+00*pb],
[[[4.2500E+02*GeV,2.5000E+01*GeV],[4.2500E+02*GeV,2.5000E+01*GeV]],1.3481E+00*pb],
[[[4.2500E+02*GeV,5.0000E+01*GeV],[4.2500E+02*GeV,5.0000E+01*GeV]],1.7300E+00*pb],
```

As we can see, the data grid is given as a Python array with the structure: [[masses, upper limit], [masses, upper limit], ...]. For prompt analyses, the relevant parameters are usually the BSM masses, since all decays are assumed to be prompt. On the other hand, results for long-lived or meta-stable particles may depend on the BSM widths as well. The width dependence can be easily included through the following generalization:

$$[[M_1, M_2...], [M_A, M_B, ...]] \rightarrow [[(M_1, \Gamma_1), (M_2, \Gamma_2)...], [(M_A, \Gamma_A), (M_B, \Gamma_B), ...]]$$

In order to make the notation more compact, whenever the width dependence is not included, the corresponding decay will be assumed to be prompt and an effective *lifetime reweigthing factor* will be applied to the upper limits. For instance, a *mixed type* data grid is also allowed:

$$[\ [[M_1, (M_2, \Gamma_2)], [M_1, (M_2, \Gamma_2)]], \text{UL}\ ], \quad [\ [[M_1', (M_2', \Gamma_2')], [M_1', (M_2', \Gamma_2')]], \text{UL'}\ ], \quad ...$$

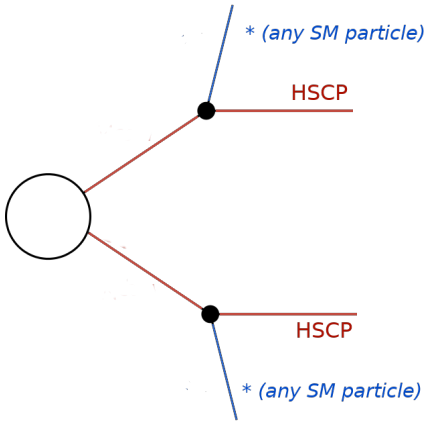
The example above represents a simplified model where the decay of the mother is prompt, while the daughter does not have to be stable, hence the dependence on Γ_2. In this case, the *lifetime reweigthing factor* is applied only for the mother decay.

Inclusive Simplified Models

If the analysis signal efficiencies or upper limits are insensitive to some of the simplified model final states, it might be convenient to define *inclusive* simplified models. A typical case are some of the heavy stable charged particle searches, which only rely on the presence of a non-relativistic charged particle, which leads to an anomalous charged track signature. In this case the signal efficiencies are highly insensitive to the remaining event activity and the corresponding simplified models can be very inclusive. In order to handle this inclusive cases in the database we allow for wildcards when specifying the constraints. For instance, the constraint for the CMS-EXO-13-006 eff/c000/THSCPM3.txt reads:

```
txName: THSCPM3
constraint: [['*']],[['*']]]
```

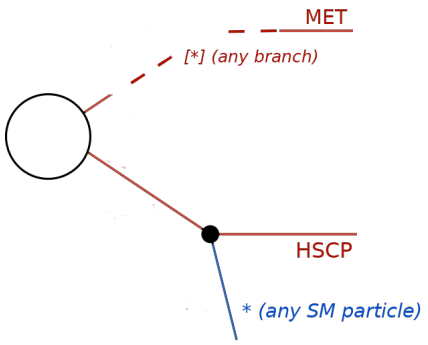
and represents the (inclusive) simplified model:



Note that although the final state represented by "*" is any Z_2-even *particle*, it must still correspond to a single particle, since the topology specifies a 2-body decay for the initially produced BSM particle. Finally, it might be useful to define even more inclusive simplified models, such as the one in CMS-EXO-13-006 eff/c000/THSCPM4.txt:

```
txName: THSCPM4
constraint: [[*],[['*']]]
finalState: ['MET', 'HSCP']
```

In the above case the simplified model corresponds to an HSCP being initially produced in association with any BSM particle which leads to a MET signature. Note that "[*]" corresponds to *any branch*, while ["*"] means *any particle*:

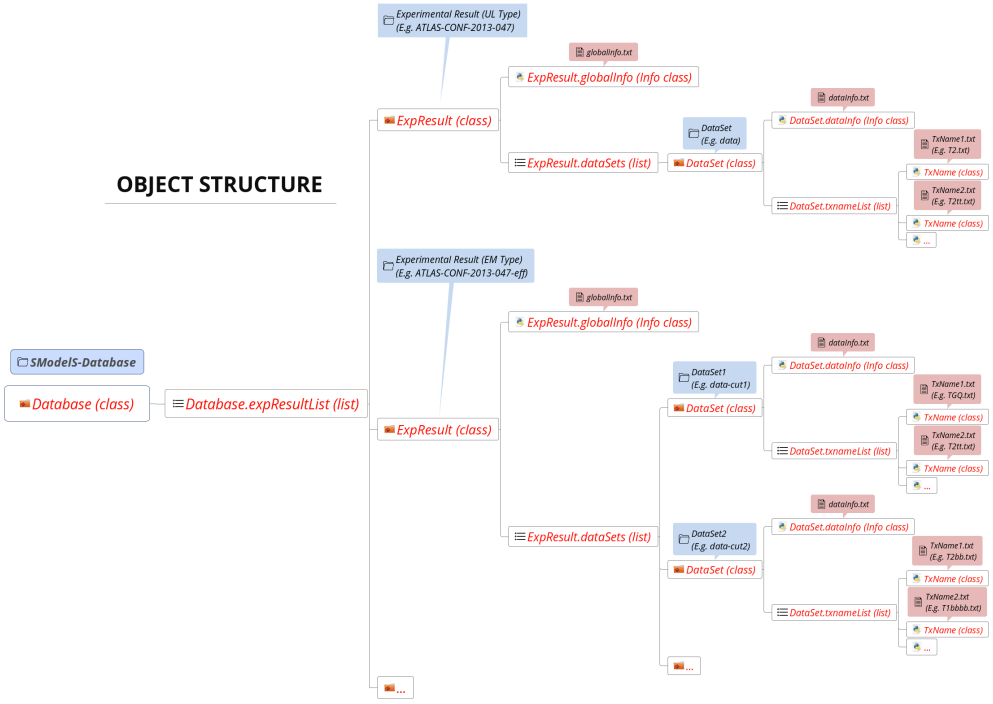


In such cases the mass array for the arbitrary branch must also be specified as using wildcards:

```
efficiencyMap: [[['*',[5.5000E+01*GeV,5.0000E+01*GeV]],5.27e-06],
[['*',[1.5500E+02*GeV,5.0000E+01*GeV]],1.28e-07],
[['*',[1.5500E+02*GeV,1.0000E+02*GeV]],0.13],
```

Database: Object Structure

The *Database folder structure* is mapped to Python objects in SModelS. The mapping is almost one-to-one, with a few exceptions. Below we show the overall object structure as well as the folders/files the objects represent (click to enlarge):



The type of Python object (Python class, Python list,...) is shown in brackets. For convenience, below we explicitly list the main database folders/files and the Python objects they are mapped to:

- *Database* folder → Database Class
- *Experimental Result* folder → ExpResult Class
- *DataSet* folder → DataSet Class
- `globalInfo.txt` file → Info Class
- `dataInfo.txt` file → Info Class
- `Txname.txt` file → TxName Class

Database: Binary (Pickle) Format

At the first time of instantiating the Database class, the text files in <*database-path*> are loaded and parsed, and the corresponding data objects are built. The efficiency and upper limit maps themselves are subjected to standard preprocessing steps such as a principal component analysis and Delaunay triangulation (see *below*). For the sake of efficiency, the entire database – including the Delaunay triangulation – is then serialized into a pickle file (<*database-path*>*/database.pcl*), which will be read directly the next time the database is loaded. If any changes in the database folder structure are detected, the python or the SModelS version has changed, SModelS will automatically re-build the pickle file. This action may take a few minutes, but it is again performed only once. If desired, the pickling process can be skipped using the option *force_load = 'txt'* in the constructor of Database .

- The pickle file is created by the createBinaryFile method

Database: Data Processing

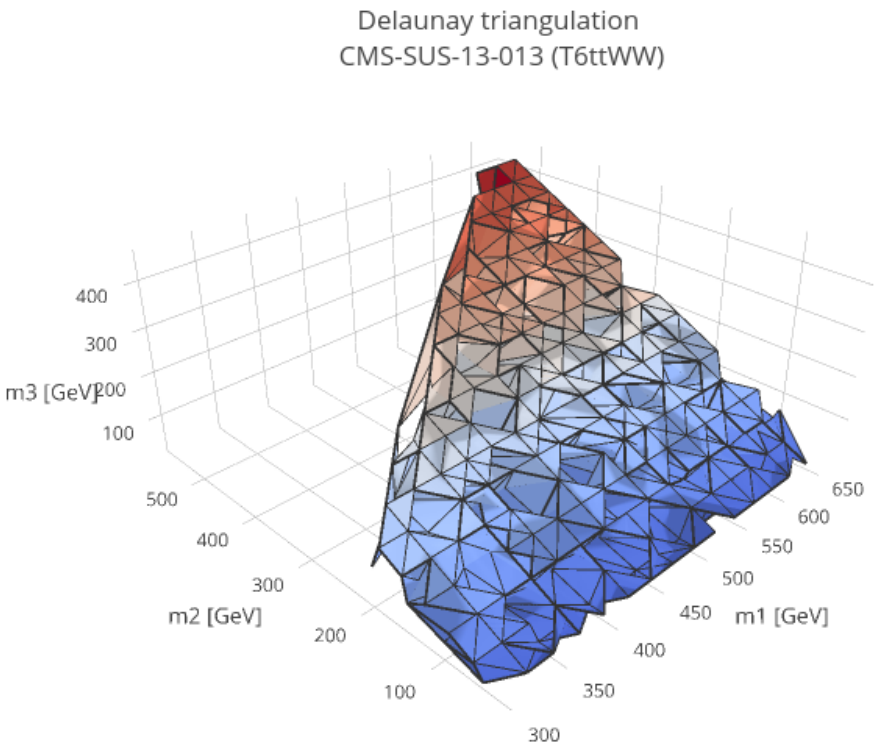
All the information contained in the *database files* is stored in the *database objects*. Within SModelS the information in the *Database* is mostly used for constraining the simplified models generated by the *decomposition* of the input model. Each simplified model (or *element*) generated is compared to the simplified models contrained by the database

and specified by the *constraint* and *finalStates* entries in the *TxName files*. The comparison allows to identify which results can be used to test the input model. Once a matching result is found the upper limit or efficiency must be computed for the given input *element*. As *described above*, the upper limits or efficiencies are provided as function of masses and widths in the form of a discrete grid. In order to compute values for any given input *element*, the data has to be processed as decribed below.

The efficiency and upper limit maps are subjected to a few standard preprocessing steps. First all the units are removed, the shape of the grid is stored and the relevant width dependence is identified (see *discussion above*). Then the masses and widths are transformed into a flat array:

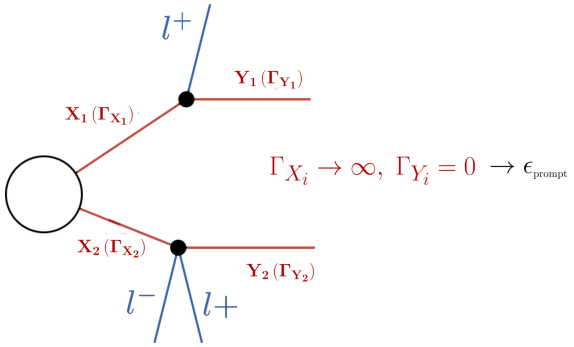
$$[[M_1, (M_2, \Gamma_2)], [M_A, (M_B, \Gamma_B)]] \to [M_1, M_2, M_A, M_B, \log(1+\Gamma_2), \log(1+\Gamma_B)]$$

Finally a principal component analysis and Delaunay triangulation (see *figure below*) is applied over the new coordinates. The simplices defined during triangulation are then used for linearly interpolating the transformed data grid, thus allowing SModelS to compute efficiencies or upper limits for arbitrary mass and width values (as long as they fall inside the data grid). As seen above, the width parameters are taken logarithmically before interpolation, which effectively corresponds to an exponential interpolation. If the data grid does not explicitly provide a dependence on all the widths (as in the *example above*), the computed upper limit or efficiency is then reweighted imposing the requirement of prompt decays (see *lifetime reweighting* for more details). This procedure provides an efficient and numerically robust way of dealing with generic data grids, including arbitrary parametrizations of the mass parameter space, irregular data grids and asymmetric branches.



Lifetime Reweighting

From v2.0 onwards SModelS allows to include width dependent efficiencies and upper limits. However most experimental results do not provide upper limits (or efficiencies) as a function of the BSM particles' widths, since usually all the decays are assumed to be prompt and the last BSM particle appearing in the cascade decay is assumed to be stable.[2] In order to apply these results to models which may contain meta-stable particles, it is possible to approximate the dependence on the widths for the case in which the experimental result requires all BSM decays to be prompt and the last BSM particle to be stable or decay *outside* the dector. In SModelS this is done through a reweighting factor which corresponds to the fraction of prompt decays (for intermediate states) and decays *outside* the detector (for final BSM states) for a given set of widths. For instance, asumme an *EM-type result* only provides efficiencies (ϵ_{prompt}) for prompt decays:



Then, for other values of the widths, an effective efficiency (ϵ_{eff}) can be approximated by:

$$\epsilon_{eff} = \xi \times \epsilon_{prompt}\ \text{, where } \xi = \mathcal{F}_{prompt}\left(\Gamma_{X_1}\right) \times \mathcal{F}_{prompt}\left(\Gamma_{X_2}\right) \times \mathcal{F}_{long}\left(\Gamma_{Y_1}\right) \times \mathcal{F}_{long}\left(\Gamma_{Y_2}\right)$$

In the expression above $\mathcal{F}_{prompt}(\Gamma)$ is the probability for the decay to be prompt given a width Γ and $\mathcal{F}_{long}(\Gamma)$ is the probability for the decay to take place *outside* the detector. The precise values of $\mathcal{F}_{prompt}$ and $\mathcal{F}_{long}$ depend on the relevant detector size (L), particle mass (M), boost (β) and width (Γ), thus requiring a Monte Carlo simulation for each input model. Since this is not within the spirit of the simplified model approach, we approximate the prompt and long-lived probabilities by:

$$\mathcal{F}_{long} = \exp\left(-\frac{\Gamma L_{outer}}{\langle\gamma\beta\rangle}\right) \text{ and } \mathcal{F}_{prompt} = 1 - \exp\left(-\frac{\Gamma L_{inner}}{\langle\gamma\beta\rangle}\right),$$

where L_{outer} is the approximate size of the detector (which we take to be 10 m for both ATLAS and CMS), L_{inner} is the approximate radius of the inner detector (which we take to be 1 mm for both ATLAS and CMS). Finally, we take the effective time dilation factor to be $\langle\gamma\beta\rangle = 1.3$ when computing $\mathcal{F}_{prompt}$ and $\langle\gamma\beta\rangle = 1.43$ when computing $\mathcal{F}_{long}$. We point out that the above approximations are irrelevant if Γ is very large ($\mathcal{F}_{prompt} \simeq 1$ and $\mathcal{F}_{long} \simeq 0$) or close to zero ($\mathcal{F}_{prompt} \simeq 0$ and $\mathcal{F}_{long} \simeq 1$). Only elements containing particles which have a considerable fraction of displaced decays will be sensitive to the values chosen above. Also, a precise treatment of lifetimes is possible if the experimental result (or a theory group) explicitly provides the efficiencies as a function of the widths, as *discussed above*.

The above expressions allows the generalization of the efficiencies computed assuming prompt decays to models with meta-stable particles. For *UL-type results* the same arguments apply with one important distinction. While efficiencies are reduced for displaced decays ($\xi < 1$), upper limits are enhanced, since they are roughly inversely proportional to signal efficiencies. Therefore, for *UL-type results*, we have:

$$\sigma^{UL}_{eff} = \sigma^{UL}_{prompt}/\xi$$

Finally, we point out that for the experimental results which provide efficiencies or upper limits as a function of some (but not all) BSM widths appearing in the simplified model (see the *discussion above*), the reweighting factor ξ is computed using only the widths not present in the grid.

[2] An obvious exception are searches for long-lived particles with displaced decays.

Confronting Predictions with Experimental Limits

Once the relevant signal cross sections (or *theory predictions*) have been computed for the input model, these must be compared to the respective upper limits. The upper limits for the signal are stored in the SModelS *Database* and depend on the type of *Experimental Result*: *UL-type* or *EM-type*.

In the case of a *UL-type result*, the theory predictions typically consist of a list of signal cross sections (one for each cluster) for the single *data set* (see *Theory Predictions for Upper Limit Results* for more details). Each theory prediction must then be compared to its corresponding upper limit. This limit is simply the cross section upper limit provided by the experimental publication or conference note and is extracted from the corresponding UL map (see *UL-type results*).

For *EM-type results* there is a single cluster for each *data set* (or signal region), and hence a single signal cross section value. This value must be compared to the upper limit for the corresponding signal region. This upper limit is easily computed using the number of observed and expected events for the *data set* and their uncertainties and is typically stored in the *Database*. Since most *EM-type results* have several signal regions (*data sets*), there will be one theory prediction/upper limit for each *data set*. By default SModelS keeps only the best *data set*, i.e. the one with the largest ratio $r_{\rm exp} = (\text{theory prediction})/(\text{expected limit})$. (See below for *combination of signal regions*) Thus each *EM-type result* will have a single theory prediction/upper limit, corresponding to the best *data set* (based on the expected limit). If the user wants to have access to all the *data sets*, the default behavior can be disabled by setting *useBestDataset=False* in theoryPredictionsFor (see *Example.py*).

The procedure described above can be applied to all the *Experimental Results* in the database, resulting in a list of theory predictions and upper limits for each *Experimental Result*. A model can then be considered excluded by the experimental results if, for one or more predictions, we have *theory prediction* $>$ *upper limit**[0].

- **The upper limits for a given** *UL-type result* **or** *EM-type result* **can be obtained using the** getUpperLimitFor method

Likelihood Computation

In the case of *EM-type results*, additional statistical information about the constrained model can be provided by the SModelS output. Most importantly, we can compute a likelihood, which describes the plausibility of a signal strength μ given the data D:

$$\mathcal{L}(\mu, \theta|D) = P\left(D|\mu + b + \theta\right) p(\theta)$$

Here, θ denotes the nuisance parameter that describes the variations in the signal and background contribtions due to systematic effects.

If no information about the correlation of signal regions is available (or if its usage is turned off, see *Using SModelS*), we compute a simplified likelihood for the most sensitive (a.k.a. best) signal region, i.e. the signal region with the highest $r_{\rm exp} = (\text{theory prediction})/(\text{expected limit})$, following the procedure detailed in CMS-NOTE-2017-001.

This means we assume $p(\theta)$ to follow a Gaussian distribution centered around zero and with a variance of δ^2, whereas $P(D)$ corresponds to a counting variable and is thus properly described by a Poissonian. The complete likelihood thus reads:

$$\mathcal{L}(\mu, \theta|D) = \frac{(\mu + b + \theta)^{n_{obs}} e^{-(\mu+b+\theta)}}{n_{obs}!} exp\left(-\frac{\theta^2}{2\delta^2}\right)$$

where n_{obs} is the number of observed events in the signal region. A test statistic T can now be constructed from a likelihood ratio test:

$$T = -2\ln\frac{H_0}{H_1} = -2\ln\left(\frac{\mathcal{L}(\mu = n_{\rm signal}, \theta|D)}{sup\{\mathcal{L}(\mu, \theta|D) : \mu \in \mathbb{R}^+\}}\right)$$

[0] The statistical significance of the exclusion statement is difficult to quantify exactly, since the model is being tested by a large number of results simultaneously.

As the signal hypothesis in the numerator presents a special case of the likelihood in the denominator, the Neyman-Pearson lemma holds, and we can assume T to be distributed according to a χ^2 distribution with one degree of freedom. Because H_0 assumes the signal strength of a particular model, $T = 0$ corresponds to a perfect match between that model's prediction and the measured data. $T \gtrsim 3.84$ corresponds to a 95% confidence level upper limit. While $n_{\rm obs}$, b and δ_b are directly extracted from the data set (coined *observedN*, *expectedBG* and *bgError*, respectively), $n_{\rm signal}$ is obtained from the calculation of the theory predictions. A default 20% systematical uncertainty is assumed for $n_{\rm signal}$, resulting in $\delta^2 = \delta_b^2 + (0.2 n_{\rm signal})^2$.

SModelS reports the χ^2 (T values) and likelihood *for each EM-type result*, together with the observed and expected r values. We note that in the general case analyses may be correlated, so summing up the T values will no longer follow a $\chi^2_{(n)}$ distribution. Therefore, for a conservative interpretation, only the result with the best expected limit should be used. Moreover, for a statistically rigorous usage in scans, it is recommended to check that the analysis giving the best expected limit does not wildly jump within continuous regions of parameter space that give roughly the same phenomenology.

- **The χ^2 for a given *EM-type result* is computed using the** chi2 method
- **The likelihood for a given *EM-type result* is computed using the** likelihood method

Combination of Signal Regions - Simplified Likelihood Approach

If the experiment provides information about the (background) correlations, signal regions can be combined. To this end, CMS sometimes provides a covariance matrix together with the efficiency maps. The usage of such covariance matrices is implemented in SModelS v1.1.3 onwards, following as above the simplified likelihood approach described in CMS-NOTE-2017-001.

SModelS allows for a marginalization as well as a profiling of the nuisances, with profiling being the default (an example for using marginalisation can be found in *How To's*). Since CPU performance is a concern in SModelS, we try to aggregate the official results, which can comprise >100 signal regions, to an acceptable number of aggregate regions. Here *acceptable* means as few aggregate regions as possible without loosing in precision or constraining power. The CPU time scales roughly linearly with the number of signal regions, so aggregating e.g. from 80 to 20 signal regions means gaining a factor 4 in computing time.

Under the assumptions described in CMS-NOTE-2017-001, the likelihood for the signal hypothesis when combining signal regions is given by:

$$\mathcal{L}(\mu, \theta | D) = \prod_{i=1}^{N} \frac{(\mu s_i^r + b_i + \theta_i)^{n_{obs}^i} e^{-(\mu s_i^r + b_i + \theta_i)}}{n_{obs}^i !} exp\left(-\frac{1}{2}\vec{\theta}^T V^{-1} \vec{\theta}\right)$$

where the product is over all N signal regions, μ is the overall signal strength, s_i^r the relative signal strength in each signal region and V represents the covariance matrix. Note, however, that unlike the case of a single signal region, we do not include any signal uncertainties, since this should correspond to a second order effect.

Using the above likelihood we compute a 95% confidence level limit on μ using the CL_s (CL_{sb}/CL_b) limit from the test statistic q_μ, as described in Eq. 14 in G. Cowan et al., Asymptotic formulae for likelihood-based tests. We then search for the value $CL_s = 0.95$ using the Brent bracketing technique available through the scipy optimize library. Note that the limit computed through this procedure applies to the total signal yield summed over all signal regions and assumes that the relative signal strengths in each signal region are fixed by the signal hypothesis. As a result, the above limit has to be computed for each given input model (or each *theory prediction*), thus considerably increasing CPU time.

When using *runSModelS.py*, the combination of signal regions is turned on or off with the parameter **options:combineSRs**, see *parameter file*. Its default value is *False*, in which case only the result from the best expected signal region (best SR) is reported. If *combineSRs* = *True*, both the combined result and the one from the best SR are quoted.

In the *figure below* we show the constraints on the simplified model T2bbWWoff when using the best signal region (left), all the 44 signal regions considered in CMS-PAS-SUS-16-052 (center) and the aggregated signal regions included in the SModelS database (right). As we can see, while the curve obtained from the combination of all 44 signal regions is much closer to the official exclusion than the one obtained using only the best SR. Finally, the aggregated result included in the SModelS database (total of 17 aggregate regions) comes with little loss in constraining power, although it considerable reduces the running time.

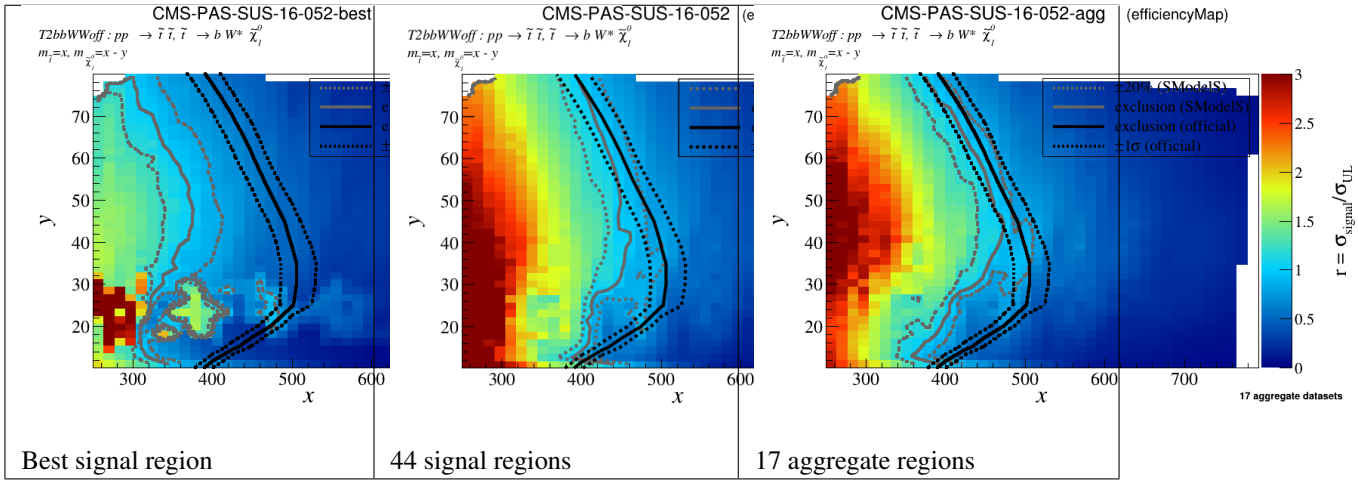


Figure: Comparison of exclusion curves for CMS-PAS-SUS-16-052 using only the best signal region (left), the combination of 17 aggregate signal regions (center), and the combination of all 44 signal regions (right).

Combination of Signal Regions - Full Likelihoods (pyhf)

In early 2020, following ATL-PHYS-PUB-2019-029, ATLAS has started to provide *full likelihoods* for results with full Run 2 luminosity (139/fb), using a JSON serialisation of the likelihood. This JSON format describes the HistFactory family of statistical models, which is used by the majority of ATLAS analyses. Thus background estimates, changes under systematic variations, and observed data counts are provided at the same fidelity as used in the experiment.

SModelS supports the usage of these JSON likelihoods from v1.2.4 onward via an interface to the pyhf package, a pure-python implementation of the HistFactory statistical model. This means that for *EM-type result* from ATLAS, for which a JSON likelihood is available and when the combination of signal regions is turned on, the evaluation of the likelihood is relegated to pyhf. Internally, the whole calculation is based on the asymptotic formulas of *Asymptotic formulae for likelihood-based tests of new physics*, arXiv:1007.1727.

The *figure below* examplifies how the constraints improve from using the best signal region (left) to using the full likelihood (right).

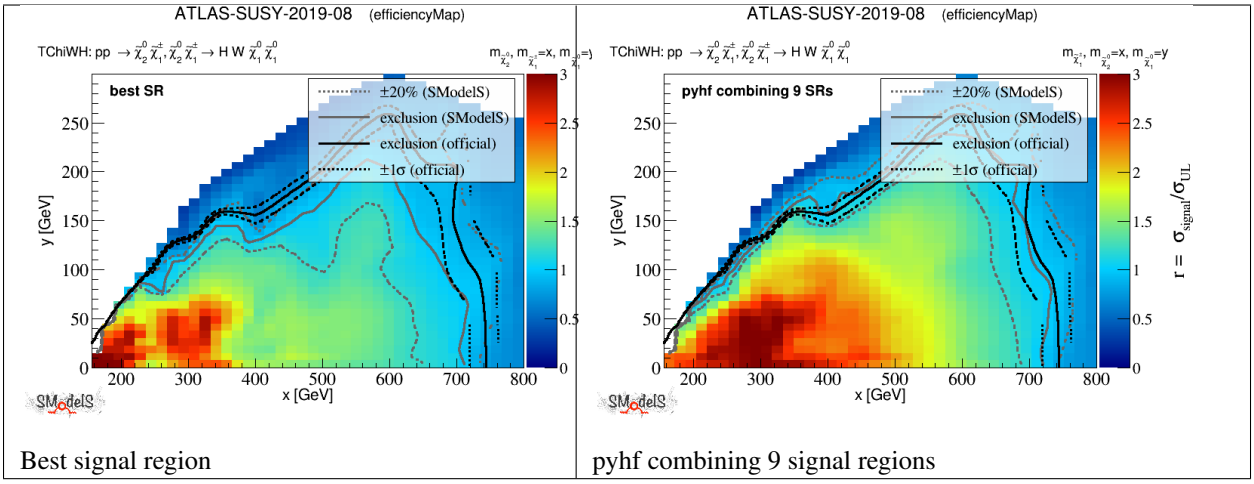


Figure: Comparison of exclusion curves for ATLAS-SUSY-2019-08 using only the best signal region (left), and the combination of all 9 signal regions with pyhf (right).

Simplified Model Coverage

The constraints provided by SModelS are obviously limited by its *database* and the available set of simplified model interpretations provided by the experimental collaborations or computed by theory groups. Therefore it is interesting to identify classes of missing simplified models (or missing topologies) which are relevant for a given input model, but are not constrained by the SModelS *database*. This task is performed as a last step in SModelS, once the *decomposition* and the *theory predictions* have been computed.

During the computation of the *theory predictions*, each *element* from the *decomposition* which matches at least one of the simplified models in the *database* is marked as "covered by" the corresponding type of *Experimental Result*. Currently the *Experimental Results* are either of type *prompt* or *displaced*.[1] If the same *element* is *covered* by both types of *Experimental Results*, it will be marked as covered by displaced *and* prompt results. If, in addition to being covered, the *element* also has a non-zero efficiency or upper limit (i.e. its properties fall inside the data grid for any result), it will be marked as "tested by" the corresponding type of result (*prompt* or *displaced*). Hence, after the *theory predictions* have been computed, the *elements* store information about their experimental coverage and can be classified and group into *coverage*.

- **The coverage tool is implemented by the** Uncovered class

Coverage Groups

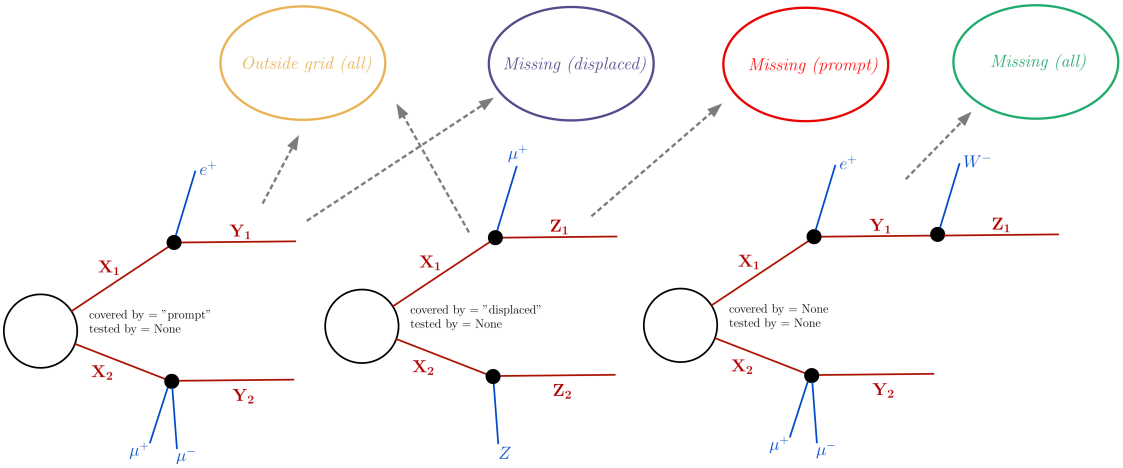
The coverage algorithm groups all the *elements* into *coverage groups* which can be easily defined by the user (see the coverage module). Each group must define a criterium for selecting *elements* after the *theory predictions* have been computed. The default *coverage groups* implemented in SModelS are:

- *missing (prompt)*: **not** covered by prompt-type results. This group corresponds to all *elements* which did not match any of the simplified models contrained by *prompt Experimental Results*.
- *missing (displaced)*: **not** covered by displaced-type results. This group corresponds to all *elements* which did not match any of the simplified models contrained by *displaced Experimental Results*.

[1] Prompt results are all those which assumes all decays to be prompt and the last BSM particle to be stable (or decay outside the detector). Searches for heavy stable charged particles (HSCPs), for instance, are classified as *prompt*, since the HSCP is assumed to decay outside the detector. Displaced results on the other hand require at least one decay to take place inside the detector.

- *missing (all)*: **not** covered by any type of result. This group corresponds to all *elements* which did not match any of the simplified models contrained by the *database*.
- *outsideGrid (all)*: covered by at least one type of *Experimental Result* and **not** tested by any type of result. This group corresponds to all *elements* which matched at least one the simplified models constrained by the *database*, but were not tested (e.g. their masses and/or widths fall outside the efficiency or upper limit grids).

The *figure below* schematically represents the grouping performed in coverage:



Besides defining which *elements* should be selected, each coverage group can also specify a reweighting function for the *element*'s cross section. This is useful for the cases where the coverage group aims to represent missing topologies with prompt (or displaced) decays, so only the fraction of prompt (displaced) cross section should be extracted. The reweighting functions defined will be applied to the selected *elements* in order to extract the desirable fraction of signal cross section for the group. For instance, for the default groups listed above, the following reweighint functions are defined:

- *missing (prompt)*: $\sigma \to \xi \times \sigma, \;\; \xi = \prod_{i=1,N-2} \mathcal{F}^i_{prompt} \times \prod_{i=N-2,N} \mathcal{F}^i_{long}$
- *missing (displaced)*: $\sigma \to \xi \times \sigma, \;\; \xi = \mathcal{F}_{displaced}(any) \times \prod_{i=N-2,N} \mathcal{F}^i_{long}$
- *missing (all)*: $\sigma \to \xi \times \sigma, \;\; \xi = 1$
- *outsideGrid (all)*: $\sigma \to \xi \times \sigma, \;\; \xi = 1$

The definition for the fraction of long-lived ($\mathcal{F}_{long}$) and prompt ($\mathcal{F}_{prompt}$) decays can be found in *lifetime reweighting*. The fraction $\mathcal{F}_{displaced}(any)$ corresponds to the probability of at least one displaced decay taking place, where the probability for a displaced decay is given by $1 - \mathcal{F}_{long} - \mathcal{F}_{prompt}$.

If *mass* or *invisible compression* are turned on, *elements* generated by *compression* and their ancestors (original/uncompressed *element*) could both fall into the same coverage group. Since the total missed cross section in a given group should equal the total signal cross section not covered or tested by the corresponding type of *Experimental Results*, one has to avoid double counting *elements*. In addition, a compressed *element* belonging to a given coverage group could combine cross sections from more than one uncompressed (original) *element*. If one of the original *elements* do not belong to this coverage group (i.e. it has been covered and/or tested by the *Experimental Results*), its contribution to the compressed *element* cross section should be subtracted. SModelS deals with the above issues through the following steps:

- an effective "missing cross section" is computed for each *element*, which corresponds to the *element* weight subtracted of the weight of its ancestors which do not belong to the same coverage group. The effective cross section also includes the reweighting *discussed above*.
- All *elements* belonging to the same group which have a common ancestor are removed (only the *element* with largest missing cross section is kept).

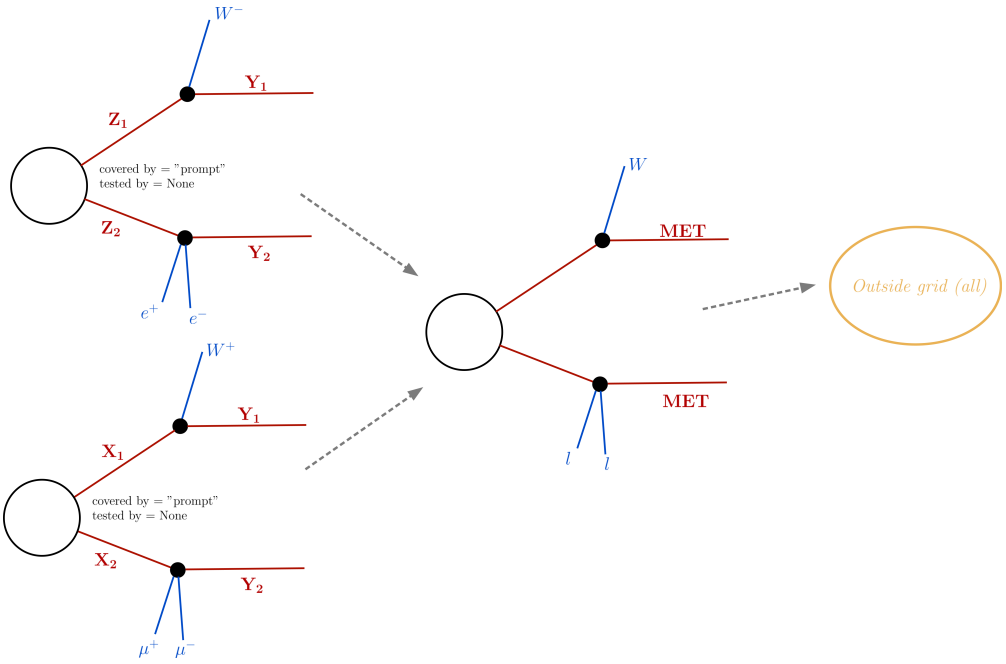
Usually the list of *elements* in each group can be considerably long, due to distinct intermediate BSM states and final SM states. In order to make the list more compact, all *elements* are further combined according to their *topology* and final state *particles* (i.e. all properties of intermediate BSM states are ignored). By default, the SM *particles* are grouped according to the definitions below:

- $W^+, W^- \rightarrow$ W
- $\tau^+, \tau^- \rightarrow$ ta
- $e^+, e^-, \mu^+, \mu^- \rightarrow$ l
- $t, \bar{t} \rightarrow$ t
- $u, d, s, c, \bar{u}, \bar{d}, \bar{s}, \bar{c}, g, pions \rightarrow$ jet
- $\nu_e, \nu_\mu, \nu_\tau, \bar{\nu}_e, \bar{\nu}_\mu, \bar{\nu}_\tau \rightarrow$ nu

while the final (Z_2-odd) BSM *particles* are grouped by their signature:

- *color and electrically neutral states* $\rightarrow$ MET
- *color neutral states with electric charge +-1* $\rightarrow$ HSCP
- *color triplet states with electric charge +-2/3 or +-1/3* $\rightarrow$ RHadronQ
- *color octates states with zero electric charge* $\rightarrow$ RHadronG

The *figure below* schematically represents the combination of elements according to the grouping of similar final states:



- **Coverage groups are implemented by the** UncoveredGroup class

In the following we discuss each of these in more detail.

Output Description

A detailed description of the possible output formats generated by *running SModelS* and their content is given below. For simplicity we will assume that all printer options in the *parameters file* are set to True, so the output information

is maximal.[1]

Screen (Stdout) Output

The stdout (or *log output*) is intended to provide extensive information about the *database*, the *decomposition*, the *theory predictions* and the *missing topologies*. It is most convenient if the input is a single file and not a folder, since the output is quite extensive. If all the options in **stdout-printer** are set to True (see *parameters file*), the screen output contains the following information:

- information about the basic input parameters and the status of the run:

```
Input status: 1
Decomposition output status: 1 #decomposition was successful
# Input File: inputFiles/slha/gluino_squarks.slha
# maxcond = 0.2
# minmassgap = 5.
# ncpus = 1
# sigmacut = 0.01
# Database version: 2.0.0
```

- a list of all the *experimental results* considered (if **printDatabase** = True). Note that this list corresponds to all the results selected in the **database** options (see *parameters file*). If **addAnaInfo** = True, for each *experimental result* entry a list of all the simplified models (or *elements*) constrained by the analysis is also shown using the *bracket notation*:

```
   ========================================================
 ||                                                      ||
 ||                  Selected Experimental Results               ||
 ||                                                      ||
   ========================================================
============================================================
Experimental Result ID: ATLAS-SUSY-2015-01
Tx Labels: ['T2bb']
Sqrts: 1.30E+01 [TeV]
         ----------------------------
         Elements tested by analysis:
            [[[b]],[[b]]] (MET, MET)
============================================================
Experimental Result ID: ATLAS-SUSY-2013-08
Tx Labels: ['T6ZZtt']
Sqrts: 8.00E+00 [TeV]
         ----------------------------
         Elements tested by analysis:
            [[[Z],[t]],[[Z],[t]]] (MET, MET)
============================================================
```

- a full list of the *topologies* generated by the *decomposition* (if **printDecomp** = True). Each *topology* entry contains basic information about the *topology* as well as the number of *elements* with this *topology* and the sum over all the *elements* weights. If **addElementInfo** = True, the *elements* belonging to each *topology* are also explicitly shown, as well as the *element*'s mass, BSM and SM final state *particles*, weight, the PIDs of the BSM *particles* contributing to the *element* and the element ID:

```
   ========================================================
 ||                                                      ||
```

(continues on next page)

[1] Some of the output may change depending on the code and database versions used.

(continued from previous page)

```
 ||                  Topologies Table                              ||
 ||                                                          ||
   ========================================================
==========================================================
Topology:
Number of vertices: [0, 0]
Number of vertex parts: [[], []]
Total Global topology weight :
Sqrts: 8.00E+00 [TeV], Weight:4.81E-04 [pb]
Sqrts: 1.30E+01 [TeV], Weight:1.58E-03 [pb]

Total Number of Elements: 1
                 ..........................................................................
↪....
                 Element:
                 Element ID: 1
                 Particles in element: [[], []]
                 Final states in element: [N1, N1]
                 The element masses are
                 Branch 0: [1.29E+02 [GeV]]
                 Branch 1: [1.29E+02 [GeV]]

                 The element PIDs are
                 PIDs: [1000022]
                 PIDs: [1000022]
                 The element weights are:
                  Sqrts: 8.00E+00 [TeV], Weight:4.81E-04 [pb]
                  Sqrts: 1.30E+01 [TeV], Weight:1.58E-03 [pb]

==========================================================
Topology:
Number of vertices: [0, 1]
Number of vertex parts: [[], [1]]
Total Global topology weight :
Sqrts: 8.00E+00 [TeV], Weight:1.16E-03 [pb]
Sqrts: 1.30E+01 [TeV], Weight:3.71E-03 [pb]

Total Number of Elements: 7
                 ..........................................................................
↪....
                 Element:
                 Element ID: 2
                 Particles in element: [[], [[W+]]]
                 Final states in element: [N1, N1]
                 The element masses are
                 Branch 0: [1.29E+02 [GeV]]
                 Branch 1: [2.69E+02 [GeV], 1.29E+02 [GeV]]

                 The element PIDs are
                 PIDs: [1000022]
                 PIDs: [1000024, 1000022]
                 The element weights are:
                  Sqrts: 8.00E+00 [TeV], Weight:2.40E-04 [pb]
                  Sqrts: 1.30E+01 [TeV], Weight:2.63E-04 [pb]

                 ..........................................................................
↪....
```

(continues on next page)

(continued from previous page)

```
                Element:
                Element ID: 3
                Particles in element: [[], [[W-]]]
                Final states in element: [N1, N1~]
                The element masses are
                Branch 0: [1.29E+02 [GeV]]
                Branch 1: [2.69E+02 [GeV], 1.29E+02 [GeV]]

                The element PIDs are
                PIDs: [1000022]
                PIDs: [-1000024, -1000022]
                The element weights are:
                 Sqrts: 8.00E+00 [TeV], Weight:4.01E-05 [pb]
```

- a list of all the *theory predictions* obtained and the corresponding *experimental result* upper limit. For each *experimental result*, the corresponding *id*, signal region (*data set*) and *sqrts* as well as the constrained simplified models (*txnames*) are printed. After this basic information, the signal cross section (*theory prediction*), the list of *condition values* (if applicable) and the corresponding observed upper limit are shown. Also, if available, the expected upper limit is included. If **computeStatistics** = True, the χ^2 and likelihood values are printed (see *likelihood calculation*). Finally, if **printExtendedResults** = True, basic information about the *elements* being constrained, such as their masses[2], IDs and PIDs, is also shown.

```
   ======================================================
 ||                                                      ||
 ||          Theory Predictions and                     ||
 ||          Experimental Constraints                      ||
 ||                                                      ||
   ======================================================

---------------Analysis Label = CMS-SUS-16-036
------------------Dataset Label = (UL)
------------------Txname Labels = ['T2']
Analysis sqrts: 1.30E+01 [TeV]
Theory prediction: 1.17E-02 [pb]
Theory conditions:  None
Observed experimental limit: 7.68E+00 [fb]
Observed r-Value: 1.5212733123965427
Masses in branch 0: [9.91E+02 [GeV], 1.29E+02 [GeV]]
Masses in branch 1: [9.91E+02 [GeV], 1.29E+02 [GeV]]
Contributing elements: [28, 29, 30, 34, 35, 36, 37, 38, 39, 40]
PIDs:-2000004
PIDs:2000004
PIDs:[-2000004, 2000004]
PIDs:[2000002, -2000002]
PIDs:[2000004, -2000004]
PIDs:2000001
PIDs:[-2000002, 2000002]
PIDs:[-1000002, 1000002]
PIDs:[-2000003, -2000001, 2000001, 2000003]
PIDs:2000002
PIDs:[-1000001, -1000003, 1000001]
PIDs:1000002
PIDs:[2000001, -2000001]
```

(continues on next page)

[2] The mass shown corresponds to an average mass over all the *elements* contributing to the theory prediction. In the case where the *elements* have distinct *topologies* no mass is shown.

(continued from previous page)

```
PIDs:[-2000003, 2000001, -2000001]
PIDs:[2000003, 2000001]
PIDs:1000001

---------------Analysis Label = CMS-SUS-16-033
-------------------Dataset Label = (UL)
-------------------Txname Labels = ['T2']
Analysis sqrts: 1.30E+01 [TeV]
Theory prediction: 1.17E-02 [pb]
Theory conditions:  None
Observed experimental limit: 8.64E+00 [fb]
Observed r-Value: 1.3528264298469297
Masses in branch 0: [9.91E+02 [GeV], 1.29E+02 [GeV]]
Masses in branch 1: [9.91E+02 [GeV], 1.29E+02 [GeV]]
Contributing elements: [28, 29, 30, 34, 35, 36, 37, 38, 39, 40]
PIDs:-2000004
PIDs:2000004
PIDs:[-2000004, 2000004]
PIDs:[2000002, -2000002]
PIDs:[2000004, -2000004]
PIDs:2000001
PIDs:[-2000002, 2000002]
PIDs:[-1000002, 1000002]
PIDs:[-2000003, -2000001, 2000001, 2000003]
PIDs:2000002
PIDs:[-1000001, -1000003, 1000001]
PIDs:1000002
PIDs:[2000001, -2000001]
PIDs:[-2000003, 2000001, -2000001]
PIDs:[2000003, 2000001]
PIDs:1000001
```

- summary information about the *missing topologies*, if **testCoverage** = True. The total missing topology cross section corresponds to the sum of cross sections of all *elements* which are not tested by any *experimental result*. The cross section for missing topologies with prompt (displaced) decays corresponds to the total signal cross section going into prompt (displaced) decays which are not tested by any displaced *experimental result* of type *prompt* (*displaced*) (see *simplified model coverage* for more details). If the *element* is constrained by one or more *experimental results*, but its masses and/or widths are outside the efficiency or upper limit grids (see *EM-type results* and *UL-type results*), its cross section is included in the total cross section outside the grid.

```
Total cross-section for missing topologies (fb):  2.742E+03
Total cross-section for missing topologies with displaced decays (fb):  0.000E+00
Total cross-section for missing topologies with prompt decays (fb):  2.742E+03
Total cross-section for topologies outside the grid (fb):  1.658E+00
```

- detailed information about the missing topologies with highest cross sections. The *element* cross section (weight) as well as the description of its final states (in *bracket notation*) is included for each distinct *coverage group*. If **addCoverageID** = True, all the *elements* IDs contributing to the missing topology are shown. These IDs can be traced back to the corresponding *elements* using the *decomposition* information obtained with **printDecomp** = True and **addElementInfo** = True.

```
================================================================================
missing topologies with the highest cross sections (up to 10):
Sqrts (TeV)   Weight (fb)                 Element description
 13.0          1.482E+02    #             [[[jet],[W]],[[jet,jet],[W]]]   (MET,MET)
Contributing elements [515, 516, 517, 518, 553, 554, 555, 556, 501, 502, 503, 504,␣
↪564, 565, 566, 567, 574, 575, 576, 577, 584, 585, 586, 587]
```

(continues on next page)

(continued from previous page)

- information about the missing topologies with displaced decays. If **addCoverageID** = True, all the *elements* IDs contributing to the missing topology are shown.

```
================================================================================
No missing topologies with displaced decays found
```

- information about the missing topologies with prompt decays. If **addCoverageID** = True, all the *elements* IDs contributing to the missing topology are shown.

```
================================================================================
missing topologies with prompt decays with the highest cross sections (up to 10):
Sqrts (TeV)   Weight (fb)                  Element description
 13.0          1.482E+02    #              [[[jet],[W]],[[jet,jet],[W]]]   (MET,MET)
Contributing elements [515, 516, 517, 518, 553, 554, 555, 556, 501, 502, 503, 504,␣
↪564, 565, 566, 567, 574, 575, 576, 577, 584, 585, 586, 587]
```

- detailed information about the topologies which are outside the *experimental results* grid. If **addCoverageID** = True, all the *elements* IDs contributing to the missing topology are shown.

```
topologies outside the grid with the highest cross sections (up to 10):
Sqrts (TeV)   Weight (fb)                  Element description
 13.0          1.371E+00    #                      [[[t],[W]],[[t],[W]]]   (MET,MET)
Contributing elements [375]
 13.0          2.871E-01    #              [[[b],[higgs]],[[b],[higgs]]]   (MET,MET)
Contributing elements [368]
================================================================================
```

Log Output

The log-type output is identical to the *screen output*, except that it is redirected to a .log file. The filename is set as the <input file>.log and stored in the output folder (see the *runSModelS options*).

Summary File Output

The summary-type output is similar to the *screen output*, but restricted to the list of *theory predictions* and model *coverage*. The output is printed to the file <input file>.smodels and stored in the output folder (see the *runSModelS options*).

Below we describe in detail the blocks contained in the summary output:

- information about the basic input parameters and the status of the run:

```
Input status: 1
Decomposition output status: 1 #decomposition was successful
# Input File: inputFiles/slha/gluino_squarks.slha
# maxcond = 0.2
# minmassgap = 5.
# ncpus = 1
# sigmacut = 0.1
# Database version: 2.0.0
```

- a list of all the *theory predictions* obtained and the corresponding *experimental result* upper limit. If **expandedSummary** = False only the most constraining *experimental result* is printed. For each *theory prediction* entry,

the corresponding *experimental result id*, the signal region (*data set*) used (only for *EM-type results*) and the *experimental result sqrts* is printed. Furthermore, the *txnames* contributing to the signal cross section, the theory cross section (*Theory_Value*), the observed upper limit (*Exp_limit*), the (theory cross section)/(observed upper limit) ratio (*r*) and, when available, the (theory cross section)/(expected upper limit) ratio (*r_expect*) are also printed. For *UL-type results* the condition violation (see *upper limit conditions*) is also included. Finally, if **computeStatistics** = True, the χ^2 and likelihood values (for *EM-type results*) are printed:

```
#Analysis  Sqrts  Cond_Violation  Theory_Value(fb)  Exp_limit(fb)  r  r_expected

     CMS-SUS-16-036  1.30E+01    0.0  1.103E+01  7.684E+00  1.435E+00  N/A
 Signal Region:  (UL)
 Txnames:  T2
--------------------------------------------------------------------------------
     CMS-SUS-16-033  1.30E+01    0.0  1.103E+01  8.639E+00  1.277E+00  N/A
 Signal Region:  (UL)
 Txnames:  T2
--------------------------------------------------------------------------------
     CMS-SUS-16-033  1.30E+01    0.0  7.530E+00  6.777E+00  1.111E+00  N/A
 Signal Region:  (UL)
 Txnames:  T1
--------------------------------------------------------------------------------
 ATLAS-SUSY-2013-02  8.00E+00    0.0  6.031E-01  1.818E+00  3.317E-01  3.988E-01
 Signal Region:  SR2jt
 Txnames:  T1, T2
 Chi2, Likelihood =  7.675E-02  1.327E-03
--------------------------------------------------------------------------------
```

- the maximum value for the (theory cross section)/(observed upper limit) ratio. If this value is higher than *1* the input model is likely excluded by one of the *experimental results* (see *confronting predictions*)

```
================================================================================
The highest r value is = 1.43531652263383
```

- summary information about the *missing topologies*, if **testCoverage** = True. The total missing topology cross section corresponds to the sum of cross sections of all *elements* which are not tested by any *experimental result*. The cross section for missing topologies with prompt (displaced) decays corresponds to the total signal cross section going into prompt (displaced) decays which are not tested by any displaced *experimental result* of type *prompt* (*displaced*) (see *simplified model coverage* for more details). If the *element* is constrained by one or more *experimental results*, but its masses and/or widths are outside the efficiency or upper limit grids (see *EM-type results* and *UL-type results*), its cross section is included in the total cross section outside the grid.

```
Total cross-section for missing topologies (fb):  2.742E+03
Total cross-section for missing topologies with displaced decays (fb):  0.000E+00
Total cross-section for missing topologies with prompt decays (fb):  2.742E+03
Total cross-section for topologies outside the grid (fb):  1.658E+00
```

- detailed information about the missing topologies with highest cross sections. The *element* cross section (weight) as well as the description of its final states (in *bracket notation*) is included for each distinct *coverage group*.

```
Full information on unconstrained cross sections
================================================================================
missing topologies with the highest cross sections (up to 10):
Sqrts (TeV)   Weight (fb)                  Element description
 13.0          1.482E+02    #              [[[jet],[W]],[[jet,jet],[W]]]   (MET,MET)
 13.0          1.375E+02    #    [[[jet,jet],[W]],[[jet],[jet,jet],[W]]]   (MET,MET)
 13.0          1.047E+02    #              [[[jet,jet],[W]],[[b,t],[W]]]   (MET,MET)
```

(continues on next page)

(continued from previous page)

```
 13.0           8.728E+01     #                    [[[jet]],[[jet,jet],[W]]]   (MET,MET)
 13.0           8.654E+01     #       [[[jet,jet],[higgs]],[[jet,jet],[W]]]   (MET,MET)
 13.0           7.520E+01     #                    [[[jet],[W]],[[b,t],[W]]]   (MET,MET)
 13.0           6.977E+01     #         [[[jet,jet],[W]],[[jet],[b,t],[W]]]   (MET,MET)
 13.0           6.977E+01     #         [[[b,t],[W]],[[jet],[jet,jet],[W]]]   (MET,MET)
 13.0           6.186E+01     #           [[[jet],[W]],[[jet,jet],[higgs]]]   (MET,MET)
 13.0           6.169E+01     #           [[[jet],[higgs]],[[jet,jet],[W]]]   (MET,MET)
```

- information about the missing topologies with displaced decays.

```
================================================================================
No missing topologies with displaced decays found
```

- information about the missing topologies with prompt decays.

```
================================================================================
missing topologies with prompt decays with the highest cross sections (up to 10):
Sqrts (TeV)    Weight (fb)                  Element description
 13.0           1.482E+02     #              [[[jet],[W]],[[jet,jet],[W]]]   (MET,MET)
 13.0           1.375E+02     #    [[[jet,jet],[W]],[[jet],[jet,jet],[W]]]   (MET,MET)
 13.0           1.047E+02     #              [[[jet,jet],[W]],[[b,t],[W]]]   (MET,MET)
 13.0           8.728E+01     #                  [[[jet]],[[jet,jet],[W]]]   (MET,MET)
 13.0           8.654E+01     #       [[[jet,jet],[higgs]],[[jet,jet],[W]]]   (MET,MET)
 13.0           7.520E+01     #                    [[[jet],[W]],[[b,t],[W]]]   (MET,MET)
 13.0           6.977E+01     #         [[[jet,jet],[W]],[[jet],[b,t],[W]]]   (MET,MET)
 13.0           6.977E+01     #         [[[b,t],[W]],[[jet],[jet,jet],[W]]]   (MET,MET)
 13.0           6.186E+01     #           [[[jet],[W]],[[jet,jet],[higgs]]]   (MET,MET)
 13.0           6.169E+01     #           [[[jet],[higgs]],[[jet,jet],[W]]]   (MET,MET)
```

- information about the topologies which are outside the *experimental results* grid

```
================================================================================
topologies outside the grid with the highest cross sections (up to 10):
Sqrts (TeV)    Weight (fb)                  Element description
 13.0           1.371E+00     #                        [[[t],[W]],[[t],[W]]]   (MET,MET)
 13.0           2.871E-01     #              [[[b],[higgs]],[[b],[higgs]]]   (MET,MET)
================================================================================
```

Python Output

The Python-type output is similar to the *screen output*, however converted to a Python dictionary. If all options are set to True, it includes information about the *decomposition*, the list of *theory predictions* and *simplified model coverage*. The output is printed to the file <input file>.py and stored in the output folder (see the *runSModelS options*).

Below we describe in detail the dictionary keys and values contained in the Python dictionary output:

- information about the basic input parameters and the status of the run stored under the *OutputStatus* key:

```
smodelsOutput =
{'OutputStatus': {'sigmacut': 0.1, 'minmassgap': 5.0, 'maxcond': 0.2, 'ncpus': 1,
↪'file status': 1, 'decomposition status': 1, 'warnings': 'Input file ok', 'input␣
↪file': 'inputFiles/slha/gluino_squarks.slha', 'database version': '2.0.0', 'smodels␣
↪version': '2.0.0'},
```

- a full list of the *elements* generated by the *decomposition* (if **addElementList** = True) stored under the *Element* key. Each list entry contains basic information about the *elements*. The list can be considerably long, so it is

recommended to set **addElementList** to False, unless the *decomposition* information is required by the user.

```
 'Element': [{'ID': 1, 'Particles': '[[], []]', 'Masses (GeV)': [[129.0], [129.0]],
↪'PIDs': [[1000022], [1000022]], 'Weights (fb)': {'xsec 8.0 TeV': 0.48, 'xsec 13.0␣
↪TeV': 1.58}, 'final states': ['N1', 'N1']},
{'ID': 2, 'Particles': '[[], [[higgs]]]', 'Masses (GeV)': [[129.0], [268.9, 129.0]],
↪'PIDs': [[1000022], [1000023, 1000022]], 'Weights (fb)': {'xsec 8.0 TeV': 0.17},
↪'final states': ['N1', 'N1']},
```

- a list of all the *theory predictions* obtained for the *experimental results*, stored under the *ExptRes* key. For each list entry, the corresponding result *id*, the *experimental result* type (if *UL-type result* or *EM-type result*), the signal region (*data set* ID), the *sqrts* and luminosity, the constrained simplified models (*txnames*), the signal cross section (theory prediction), the corresponding observed upper limit and the maximum condition violation (see *upper limit conditions*) are shown. Furthermore, the masses and widths[3] of the *elements* contributing to the signal cross section, the individual contribution of each *txname* (if **addTxWeights** = True) and the χ^2 and likelihood values (if **computeStatistics** = True) are also included.

```
'ExptRes': [{'maxcond': 0.0, 'theory prediction (fb)': 11.03, 'upper limit (fb)': 7.
↪68, 'expected upper limit (fb)': None, 'TxNames': ['T2'], 'Mass (GeV)': [[991.43,␣
↪129.0], [991.3, 129.0]], 'AnalysisID': 'CMS-SUS-16-036', 'DataSetID': None,
↪'AnalysisSqrts (TeV)': 13.0, 'lumi (fb-1)': 35.9, 'dataType': 'upperLimit', 'r': 1.
↪43, 'r_expected': None, 'Width (GeV)': [['prompt', 'stable'], ['prompt', 'stable']],
↪ 'TxNames weights (fb)': {'T2': 11.03}},
```

- the total cross section in each *missing topology group* (if **testCoverage** = True) followed by a list of *elements* contributing to the group. For each missing topology (element) the missing cross section (weight), the IDs of the *elements* contributing to the topology and its description in *bracket notation* is included (see *missing topology group*).

```
'Total xsec for missing topologies (fb)': 2742.22, 'missing topologies': [{'sqrts␣
↪(TeV)': 13.0, 'weight (fb)': 148.16, 'element': '[[[jet],[W]],[[jet,jet],[W]]] ␣
↪(MET,MET)', 'element IDs': [515, 516, 517, 518, 553, 554, 555, 556, 501, 502, 503,␣
↪504, 564, 565, 566, 567, 574, 575, 576, 577, 584, 585, 586, 587]},
'Total xsec for missing topologies with displaced decays (fb)': 0.0, 'missing␣
↪topologies with displaced decays': [],
'Total xsec for missing topologies with prompt decays (fb)': 2742.22, 'missing␣
↪topologies with prompt decays': [{'sqrts (TeV)': 13.0, 'weight (fb)': 148.16,
↪'element': '[[[jet],[W]],[[jet,jet],[W]]]  (MET,MET)', 'element IDs': [515, 516,␣
↪517, 518, 553, 554, 555, 556, 501, 502, 503, 504, 564, 565, 566, 567, 574, 575, 576,
↪ 577, 584, 585, 586, 587]},
'Total xsec for topologies outside the grid (fb)': 1.66, 'topologies outside the grid
↪': [{'sqrts (TeV)': 13.0, 'weight (fb)': 1.37, 'element': '[[[t],[W]],[[t],[W]]] ␣
↪(MET,MET)', 'element IDs': [375]},
```

XML Output

The xml-type output is identical to the *python output*, however converted to a xml format. The output is printed to the file <input file>.xml and stored in the output folder (see the *runSModelS options*).

Since the output information and options are the same as described for *python output*, we simply show below an excerpt of the xml file to illustrate the output format:

[3] For particles considered with prompt decays (see *promptWidth parameter*), the width is replaced by 'prompt', while for particles considered stable (see see *stableWidth parameter*), the width is replaced by 'stable'. If more than one *element* contributes to the theory prediction, the average widths and masses of the BSM states are shown.

```
    <OutputStatus>
        <database_version>2.0.0</database_version>
        <decomposition_status>1</decomposition_status>
        <file_status>1</file_status>
        <input_file>inputFiles/slha/gluino_squarks.slha</input_file>
        <maxcond>0.2</maxcond>
        <minmassgap>5.0</minmassgap>
        <ncpus>1</ncpus>
        <sigmacut>0.1</sigmacut>
        <smodels_version>2.0.0</smodels_version>
        <warnings>Input file ok</warnings>
    </OutputStatus>
    <Element_List>
        <Element>
            <ID>1</ID>
            <Masses_GeV_List>
                <Masses_GeV_List>
                    <Masses_GeV>129.0</Masses_GeV>
                </Masses_GeV_List>
                <Masses_GeV_List>
                    <Masses_GeV>129.0</Masses_GeV>
                </Masses_GeV_List>
            </Masses_GeV_List>
            <PIDs_List>
                <PIDs_List>
                    <PIDs>1000022</PIDs>
                </PIDs_List>
                <PIDs_List>
                    <PIDs>1000022</PIDs>
                </PIDs_List>
            </PIDs_List>
            <Particles>[[], []]</Particles>
            <Weights_fb>
                <xsec_13.0_TeV>1.5801581999999998</xsec_13.0_TeV>
                <xsec_8.0_TeV>0.48082062600000003</xsec_8.0_TeV>
            </Weights_fb>
            <final_states_List>
                <final_states>N1</final_states>
                <final_states>N1</final_states>
            </final_states_List>
        </Element>
    <ExptRes_List>
        <ExptRes>
            <AnalysisID>CMS-SUS-16-036</AnalysisID>
            <AnalysisSqrts_TeV>13.0</AnalysisSqrts_TeV>
            <DataSetID>None</DataSetID>
            <Mass_GeV_List>
                <Mass_GeV_List>
                    <Mass_GeV>991.43</Mass_GeV>
                    <Mass_GeV>129.0</Mass_GeV>
                </Mass_GeV_List>
                <Mass_GeV_List>
                    <Mass_GeV>991.3</Mass_GeV>
                    <Mass_GeV>129.0</Mass_GeV>
                </Mass_GeV_List>
            </Mass_GeV_List>
    <Total_xsec_for_missing_topologies_fb>2742.2252443735856</Total_xsec_for_missing_
↪topologies_fb>
```

(continues on next page)

(continued from previous page)

```
    <Total_xsec_for_missing_topologies_with_displaced_decays_fb>0.0</Total_xsec_for_
↪missing_topologies_with_displaced_decays_fb>
    <Total_xsec_for_missing_topologies_with_prompt_decays_fb>2742.2252443735856</
↪Total_xsec_for_missing_topologies_with_prompt_decays_fb>
    <Total_xsec_for_topologies_outside_the_grid_fb>1.6578659304326262</Total_xsec_for_
↪topologies_outside_the_grid_fb>
    <missing_topologies_List>
        <missing_topologies>
            <element>[[[jet],[W]],[[jet,jet],[W]]]   (MET,MET)</element>
            <element_IDs_List>
```

SLHA Output

An SLHA-type output format is also available containing a summary of the *theory predictions* and *simplified model coverage*. The file contains the SLHA-type blocks: *SModelS_Settings*, *SModelS_Exclusion* and *SModelS_Coverage*. Below we give a description of each block together with a sample output.

- information about the main input parameters:

```
BLOCK SModelS_Settings
 0 v2.0.0                    #SModelS version
 1 2.0.0                     #database version
 2 0.2                       #maximum condition violation
 3 1                         #compression (0 off, 1 on)
 4 5.                        #minimum mass gap for mass compression [GeV]
 5 0.1                       #sigmacut [fb]
 6 0                         #signal region combination (0 off, 1 on)
```

- information about the status of the input model: excluded (1), not excluded (0) or not tested (-1):

```
BLOCK SModelS_Exclusion
 0 0 1                              #output status (-1 not tested, 0 not excluded, 1␣
↪excluded)
```

- followed by the list of experimental results. If **expandedOutput** = True, all results are printed. Otherwise, if the model is excluded, all results with r-value greater than one are shown and if the point is not excluded, only the result with the highest r-value is displayed. For each experimental result, the *Txname*, the r-value, the *condition violation* and the experimental result ID are shown. If **computeStatistics** = True, the χ^2 and likelihood values for *EM-type results* are also printed:

```
 1 0 T2                             #txname
 1 1 1.435E+00                      #r value
 1 2 1.435E+00                      #expected r value
 1 3 0.00                           #condition violation
 1 4 CMS-SUS-16-036                 #analysis
 1 5 (UL)                           #signal region
 1 6 N/A                            #Chi2
 1 7 N/A                            #Likelihood
```

- the label of each *coverage group* (first entry) and the total cross section in each group (second entry) (if **testCoverage** = True):

```
BLOCK SModelS_Coverage
 0 0 missing (all)                        # missing topologies
```

(continues on next page)

(continued from previous page)

```
 0 1 2.742E+03                          # Total cross-section (fb)
 1 0 missing (displaced)                # missing topologies with displaced decays
 1 1 0.000E+00                          # Total cross-section (fb)
 2 0 missing (prompt)                   # missing topologies with prompt decays
 2 1 2.742E+03                          # Total cross-section (fb)
 3 0 outsideGrid (all)                  # topologies outside the grid
 3 1 1.658E+00                          # Total cross-section (fb)
```

Multiple Files Summary Output

When running SModelS over multiple files it might be desirable to have a simplified output with a summary of the results for each input file. Since version 2.1 this information is stored by default in the 'summary.txt' file in the output folder. This text file contains a single line for each input file and provides basic information about the SModelS results. This information includes the most constraining analysis (the one with largest observed r), the respective expected r-value (if available) and the most sensitive ATLAS and CMS analyses (the ones with largest expected r), as shown below:

```
#Global results summary (4 files)
#The most constraining analysis corresponds to the one with largest observed r.
#The most senstive (ATLAS/CMS) analysis corresponds to the one with largest expected␣
↪r from those analyses for which this information is available.
#filename            MostConstrainingAnalysis  r_max  r_exp  MostSensitive(ATLAS) ␣
↪r(ATLAS)  r_exp(ATLAS)  MostSensitive(CMS)  r(CMS)  r_exp(CMS)
complicated.slha     ATLAS-SUSY-2016-07        7.42   8.8    ATLAS-SUSY-2016-07    7.
↪42      8.8           CMS-SUS-19-006      2.15    1.79
gluinoToTops.slha    CMS-SUS-19-006            58.1   68.1   ATLAS-SUSY-2013-18    1.
↪93      2.73          CMS-SUS-19-006      58.1    68.1
gluino_squarks.slha  ATLAS-SUSY-2016-07        7.42   8.8    ATLAS-SUSY-2016-07    7.
↪42      8.8           CMS-SUS-19-006      2.15    1.79
higgsinoStop.slha    CMS-PAS-SUS-13-023        50.1   -1     ATLAS-SUSY-2013-15    42.
↪9       39.5            CMS-PAS-SUS-13-015  25.5    29.6
```

1.6 How To's

Below we provide a few examples for using SModelS and some of the *SModelS tools* as a Python library*[0].

To try out the examples in interactive mode:

Main examples:

- How to run SModelS using a parameter file (download the Python code `here`, IPython notebook `here`)
- How to run SModelS as a python library (download the Python code `here`, IPython notebook `here`)

Examples displaying several functionalities:

- How to load the database (download the Python code `here`, IPython notebook `here`)
- How to obtain experimental upper limits (download the Python code `here`, IPython notebook `here`)

[0] Some of the output may change depending on the database version used.

- How to obtain experimental efficiencies (download the Python code here, IPython notebook here)
- How to print decomposition results (download the Python code here, IPython notebook here)
- How to print theory predictions (download the Python code here, IPython notebook here)
- How to compare theory predictions with experimental limits (download the Python code here, IPython notebook here)
- How to use a LHE input including width information (download the Python code here, IPython notebook here)
- How to compute the likelihood and chi2 for a theory predictions (download the Python code here, IPython notebook here)
- How to find missing topologies (download the Python code here, IPython notebook here)
- How to generate ascii graphs (download the Python code here, IPython notebook here)
- How to marginalize a combined limit instead of profiling it (download the Python code here, IPython notebook here)

Examples using the cross-section computer:

- How to compute leading order cross sections (for MSSM) (download the Python code here, IPython notebook here)
- How to compute next-to-leading order cross sections (for MSSM) (download the Python code here, IPython notebook here)

Examples using the Database Browser

- How to obtain upper limits (download the Python code here, IPython notebook here)
- How to select specific results (download the Python code here, IPython notebook here)

Examples using the Interactive Plots tool

- How to make interactive plots (download the Python code here, IPython notebook here)

1.7 SModelS Code Documentation

These pages constitute the SModelS code documentation.

Contents

theory package

Submodules

theory.auxiliaryFunctions module

theory.auxiliaryFunctions.**addInclusives**(*massList*, *shapeArray*)
Add entries corresponding to '*' in shapeArray. If shapeArray contains entries = '*', the corresponding entries in value will be added from the output.

Parameters

- **massList** – 1D array of floats. Its dimension should be equal to the number of non "*" items in shapeArray (e.g. [200.0,100.0])
- **shapeArray** – 1D array containing the data type and '*'. The values of data type are ignored (e.g. [float,'*',float,'*','*']).

Returns original array with '*' inserted at the correct entries.

theory.auxiliaryFunctions.**addUnit**(*obj*, *unit*)

Add unit to object. If the object is a nested list, adds the unit to all of its elements.

Parameters

- **obj** – Object without units (e.g. [[100,100.]])
- **unit** – Unit to be added to the object (Unum object, e.g. GeV)

Returns Object with units (e.g. [[100*GeV,100*GeV]])

theory.auxiliaryFunctions.**average**(*values*, *weights=None*)

Compute the weighted average of a list of objects. All the objects must be of the same type. If all objects are equal returns the first entry of the list. Only the average of ints, floats and Unum objects or nested lists of these can be computed. If the average can not be computed returns None.

Parameters

- **values** – List of objects of the same type
- **weights** – Weights for computing the weighted average. If None it will assume unit weights.

theory.auxiliaryFunctions.**cGtr**(*weightA*, *weightB*)

Define the auxiliary greater function.

Return a number between 0 and 1 depending on how much it is violated (0 = A > B, 1 = A << B).

Returns XSectioList object with the values for each label.

theory.auxiliaryFunctions.**cSim**(**weights*)

Define the auxiliar similar function.

Return the maximum relative difference between any element weights of the list, normalized to [0,1].

Returns XSectionList object with the values for each label.

theory.auxiliaryFunctions.**elementsInStr**(*instring*, *removeQuotes=True*)

Parse instring and return a list of elements appearing in instring. instring can also be a list of strings.

Parameters

- **instring** – string containing elements (e.g. "[[['e+']],[['e-']]]+[[['mu+']],[['mu-']]]")
- **removeQuotes** – If True, it will remove the quotes from the particle labels. Set to False, if one wants to run eval on the output.

Returns list of elements appearing in instring in string format

theory.auxiliaryFunctions.**flattenArray**(*objList*)

Flatten any nested list to a 1D list.

Parameters **objList** – Any list or nested list of objects (e.g. [[[(100.,1e-10),100.],1.],[[200.,200.],2.],..]

Returns 1D list (e.g. [100.,1e-10,100.,1.,200.,200.,2.,..])

theory.auxiliaryFunctions.**getAttributesFrom**(*obj*, *skipIDs=[]*)

Loops over all attributes in the object and return a list of the attributes.

Parameters

- **obj** – Any object with a __dict__ attribute
- **skipIDs** – List of object ids. Any object which has its id on the list will be ignored (useful to avoid recursion).

Returns List with unique attribute labels.

theory.auxiliaryFunctions.**getValuesForObj**(*obj*, *attribute*)

Loops over all attributes in the object and in its attributes and returns a list of values for the desired attribute:

Parameters

- **obj** – Any object with a __dict__ attribute
- **attribute** – String for the desired attribute

Returns List with unique attribute values. If the attribute is not found, returns empty list.

theory.auxiliaryFunctions.**index_bisect**(*inlist*, *el*)

Return the index where to insert item el in inlist. inlist is assumed to be sorted and a comparison function (lt or cmp) must exist for el and the other elements of the list. If el already appears in the list, inlist.insert(el) will insert just before the leftmost el already there.

theory.auxiliaryFunctions.**removeInclusives**(*massArray*, *shapeArray*)

Remove all entries corresponding to '*' in shapeArray. If shapeArray contains entries = '*', the corresponding entries in value will be removed from the output.

Parameters

- **massArray** – Array to be formatted (e.g. [[200.,100.],[200.,100.]] or [[200.,'*'],'*'],0.4])
- **shapeArray** – Array with format info (e.g. ['*',[float,float]])

Returns formatted array (e.g. [[200.,100.]] or [[200.]],0.4])

theory.auxiliaryFunctions.**removeUnits**(*value*, *standardUnits*)

Remove units from unum objects. Uses the units defined in physicsUnits.standard units to normalize the data.

Parameters

- **value** – Object containing units (e.g. [[100*GeV,100.*GeV],3.*pb])
- **standardUnits** – Unum unit or Array of unum units defined to normalize the data.

Returns Object normalized to standard units (e.g. [[100,100],3000])

theory.auxiliaryFunctions.**rescaleWidth**(*width*)

The function that is applied to all widths to map it into a better variable for interpolation. It grows logarithmically from zero (for width=0.) to a large number (machine dependent) for width = infinity.

Parameters **width** – Width value (in GeV) with or without units

Return x Coordinate value (float)

theory.auxiliaryFunctions.**reshapeList**(*objList*, *shapeArray*)

Reshape a flat list according to the shape of shapeArray. The number of elements in shapeArray should equal the length of objList.

Parameters

- **objList** – 1D list of objects (e.g. [200,100,'*',50,'*'])

- **shapeArray** – Nested array (e.g. [[float,float,'*',float],'*'])

Returns Array with elements from objList shaped according to shapeArray (e.g. [[200.,100.,'*',50],'*'])

theory.auxiliaryFunctions.**unscaleWidth**(*x*)

Maps a coordinate value back to width (with GeV unit). The mapping is such that x=0->width=0 and x=very large -> width = inf.

Parameters **x** – Coordinate value (float)

Return width Width value (in GeV) with unit

theory.branch module

class theory.branch.**Branch**(*info=None*, *finalState=None*, *intermediateState=None*, *model=None*)

Bases: object

An instance of this class represents a branch. A branch-element can be constructed from a string (e.g., ('[b,b],[W]').

Initializes the branch. If info is defined, tries to generate the branch using it.

Parameters

- **info** – string describing the branch in bracket notation (e.g. [[e+],[jet]])
- **finalState** – final state label string for the branch (e.g. 'MET' or 'HSCP')
- **intermediateState** – list containing intermediate state labels (e.g. ['gluino','C1+'])
- **model** – The model (Model object) to be used when converting particle labels to particle objects (only used if info, finalState or intermediateState != None).

copy()

Generate an independent copy of self. Faster than deepcopy.

Returns Branch object

decayDaughter()

Generate a list of all new branches generated by the 1-step cascade decay of the current branch daughter. :returns: list of extended branches (Branch objects). Empty list if daughter is stable or if daughterID was not defined.

getAverage(*attr*)

Get the average value for a given attribute appearing in the odd particles of branch.

getInfo()

Get branch topology info from evenParticles.

Returns dictionary containing vertices and number of final states information

getLength()

Returns the branch length (number of odd particles).

Returns length of branch (number of odd particles)

removeVertex(*iv*)

Remove vertex iv. The "vertex-mother" in BSMparticles and (SM) particles in the vertex are removed from the branch. The vertex index corresponds to the BSM decay (iv = 0 will remove the first BSM particle,...)

Parameters **iv** – Index of vertex in branch (int)

setInfo()
Defines the number of vertices (vertnumb) and number of even particles in each vertex (vertpats) properties, if they have not been defined yet.

class theory.branch.**InclusiveBranch**(*finalState=None*, *intermediateState=None*, *model=None*)
Bases: *theory.branch.Branch*

An inclusive branch class. It will return True when compared to any other branch object with the same final state. If intermediateState is defined, it will use all the odd (BSM) particles for comparison while neglecting even particles.

Parameters

- **info** – string describing the branch in bracket notation (e.g. [[e+],[jet]])
- **finalState** – final state label string for the branch (e.g. 'MET' or 'HSCP')
- **intermediateState** – list containing intermediate state labels (e.g. ['gluino','C1+'])
- **model** – The model (Model object) to be used when converting particle labels to particle objects (only used if info, finalState or intermediateState != None).

decayDaughter()
Always return False.

getInfo()
Get branch topology info (inclusive list and int).

Returns dictionary containing vertices and number of final states information

theory.branch.**decayBranches**(*branchList*, *sigcut=0.0*)
Decay all branches from branchList until all unstable intermediate states have decayed.

Parameters

- **branchList** – list of Branch() objects containing the initial mothers
- **sigcut** – minimum sigma*BR (in fb) to be generated, by default sigcut = 0. (all branches are kept)

Returns list of branches (Branch objects)

theory.clusterTools module

class theory.clusterTools.**AverageElement**(*elements=[]*)
Bases: smodels.theory.element.Element

Represents an element or list of elements containing only the basic attributes required for clustering or computing efficiencies/upper limits. Its properties are given by the average properties of the elements it represents and its weight is given by the total weight of all elements.

contains(*element*)
Check if the average element contains the element

Parameters **element** – Element object

Returns True/False

getAverage(*attribute*, *weighted=True*)
Compute the average value for the attribute using the elements in self.elements. If weighted = True, compute the weighted average using the elements weights.

Parameters **attribute** – Name of attribute to be averaged over (string)

Returns Average value of attribute.

class theory.clusterTools.**ElementCluster**(*elements=[]*, *dataset=None*, *distanceMatrix=None*)

Bases: object

An instance of this class represents a cluster of elements. This class is used to store the relevant information about a cluster of elements and to manipulate this information.

add(*elements*)
Add an element or list of elements.

Parameters **elements** – Element object or list of elements

averageElement()
Computes the average element for the cluster. The average element is an empty element, but with its mass and width given by the average over the cluster elements.

Returns Element object

copy()
Returns a copy of the index cluster (faster than deepcopy).

getDataType()
Checks to which type of data (efficiency map or upper limit) the cluster refers to. It uses the cluster.dataset attribute. If not defined, returns None :return: upperLimits or efficiencyMap (string)

getDistanceTo(*element*)
Return the maximum distance between any elements belonging to the cluster and element.

Parameters **element** – Element object

Returns maximum distance (float)

getTotalXSec()
Return the sum over the cross sections of all elements belonging to the cluster.

Returns sum of weights of all the elements in the cluster (XSectionList object)

indices()
Return a list of element indices appearing in cluster

isConsistent(*maxDist*)
Checks if the cluster is consistent. Computes an average element in the cluster and checks if this average element belongs to the cluster according to the maximum allowed distance between cluster elements.

Returns True/False if the cluster is/is not consistent.

remove(*elements*)
Remove an element or a list of element from the cluster.

Parameters **elements** – Element object or list of elements

theory.clusterTools.**clusterElements**(*elements*, *maxDist*, *dataset*)
Cluster the original elements according to their distance in upper limit space.

Parameters

- **elements** – list of elements (Element objects)
- **dataset** – Dataset object to be used when computing distances in upper limit space
- **maxDist** – maximum distance for clustering two elements

Returns list of clusters (ElementCluster objects)

theory.clusterTools.**doCluster**(*elements*, *dataset*, *maxDist*)

Cluster algorithm to cluster elements.

Parameters

- **elements** – list of all elements to be clustered
- **dataset** – Dataset object to be used when computing distances in upper limit space
- **maxDist** – maximum distance for clustering two elements

Returns a list of ElementCluster objects containing the elements belonging to the cluster

theory.clusterTools.**groupElements**(*elements*, *dataset*)

Group elements into groups where the average element identical to all the elements in group. The groups contain all elements which share the same mass,width and upper limit and can be replaced by their average element when building clusters.

Parameters

- **elements** – list of all elements to be grouped
- **dataset** – Dataset object to be used when computing distances in upper limit space

Returns a list of AverageElement objects which represents a group of elements with same mass, width and upper limit.

theory.clusterTools.**relativeDistance**(*el1*, *el2*, *dataset*)

Defines the relative distance between two elements according to their upper limit values. The distance is defined as d = 2*|ul1-ul2|/(ul1+ul2).

Parameters

- **el1** – Element object
- **el2** – Element object

Returns relative distance

theory.crossSection module

class theory.crossSection.**XSection**

Bases: object

An instance of this class represents a cross section.

This class is used to store the information of a single cross section (value, particle ids, center of mass, order and label).

order = 0 (LO), 1 (NLO) or 2 (NLL).

Initializes the object to store a cross section value. All initial info is set to None.

copy()

Generates an independent copy of self.

Faster than deepcopy.

niceStr()

Generates a more human readable string. The string format is: Sqrts: self.info.sqrts, Weight: self.value

pid

class theory.crossSection.**XSectionInfo**(*sqrts=None*, *order=None*, *label=None*)
Bases: object

An instance of this class represents information regarding a cross section.

This class is used to store information of a cross section (center of mass, order and label).

Constructor. :param: sqrts center of mass energy, with unit :param: order perturbation order of xsec computation :param: label, a string that describes the xsec computation

copy()
Generate an independent copy of self.

Faster than deepcopy.

class theory.crossSection.**XSectionList**(*infoList=None*)
Bases: object

An instance of this class represents a list of cross sections.

This class is used to store a list of cross sections. The list is sorted by cross section, highest cross section first.

If infoList is defined, create entries with zero cross sections according to infoList. infoList must be a list of XSectionInfo objects.

add(*newxsec*)
Append a XSection object to the list.

copy()
Generates an independent copy of itself. Faster than deepcopy.

delete(*xSec*)
Delete the cross section entry from the list.

getDictionary(*groupBy='pids'*)
Convert the list of XSection objects to a nested dictionary.

First level keys are the particles IDs (if groupBy == pids) or labels (if groupBy == labels) and values are the cross section labels or particle IDs and the cross section value.

getInfo()
Get basic info about the cross sections appearing in the list (order, value and label).

Returns list of XSectionInfo objects

getMaxXsec()
Get the maximum cross section value appearing in the list.

getMinXsec()
Get minimum cross section value appearing in the list.

getPIDpairs()
Get all particle ID pairs appearing in the list.

getPIDs()
Get all particle IDs appearing in the list.

getXsecsFor(*item*)
Return a list of XSection objects for item (label, pid, sqrts).

niceStr()

order()
Order the cross section in the list by their PDG pairs

removeDuplicates()
If there are two entries for the same process, center of mass energy and order, keep only one (the one with highest value).

removeLowerOrder()
Keep only the highest order cross section for each process in the list.

Remove order information and set default labels.

sort()
sort the xsecs by the values

theory.crossSection.**getXsecFromLHEFile**(*lhefile*, *addEvents=True*)
Obtain cross sections from input LHE file.

Parameters

- **lhefile** – LHE input file with unweighted MC events
- **addEvents** – if True, add cross sections with the same mothers, otherwise return the event weight for each pair of mothers

Returns a XSectionList object

theory.crossSection.**getXsecFromSLHAFile**(*slhafile, useXSecs=None, xsecUnit=1.00E+00 [pb]*)
Obtain cross sections for pair production of R-odd particles from input SLHA file. The default unit for cross section is pb.

Parameters

- **slhafile** – SLHA input file with cross sections
- **useXSecs** – if defined enables the user to select cross sections to use. Must be a XSecInfoList object
- **xsecUnit** – cross section unit in the input file (must be a Unum unit)

Returns a XSectionList object

theory.decomposer module

theory.decomposer.**decompose**(*model, sigmacut=1.00E-01 [fb], doCompress=True, doInvisible=True, minmassgap=0.00E+00 [GeV]*)
Perform decomposition using the information stored in model.

Parameters

- **sigmacut** – minimum sigma*BR to be generated, by default sigmacut = 0.1 fb
- **doCompress** – turn mass compression on/off
- **doInvisible** – turn invisible compression on/off
- **minmassgap** – maximum value (in GeV) for considering two R-odd particles degenerate (only revelant for doCompress=True)

Returns list of topologies (TopologyList object)

theory.element module

class theory.element.**Element**(*info=None*, *finalState=None*, *intermediateState=None*, *model=None*)

Bases: object

An instance of this class represents an element. This class possesses a pair of branches and the element weight (cross-section * BR).

Initializes the element. If info is defined, tries to generate the element using it.

Parameters

- **info** – string describing the element in bracket notation (e.g. [[[e+],[jet]],[[e-],[jet]]])
- **finalState** – list containing the final state labels for each branch (e.g. ['MET', 'HSCP'] or ['MET','MET'])
- **intermediateState** – nested list containing intermediate state labels for each branch (e.g. [['gluino'], ['gluino']])
- **model** – The model (Model object) to be used when converting particle labels to particle objects (only used if info, finalState or intermediateState != None).

checkConsistency()

Check if the particles defined in the element are consistent with the element info.

Returns True if the element is consistent. Print error message and exits otherwise.

compressElement(*doCompress*, *doInvisible*, *minmassgap*)

Keep compressing the original element and the derived ones till they can be compressed no more.

Parameters

- **doCompress** – if True, perform mass compression
- **doInvisible** – if True, perform invisible compression
- **minmassgap** – value (in GeV) of the maximum mass difference for compression (if mass difference < minmassgap, perform mass compression)

Returns list with the compressed elements (Element objects)

copy()

Create a copy of self. Faster than deepcopy.

Returns copy of element (Element object)

getAncestors()

Get a list of all the ancestors of the element. The list is ordered so the mothers appear first, then the grandmother, then the grandgrandmothers,...

Returns A list of Element objects containing all the ancestors sorted by generation.

getAverage(*attr*)

Get the average value for a given attribute appearing in the odd particles of the element branches.

getEinfo()

Get element topology info from branch topology info.

Returns dictionary containing vertices and number of final states information

getFinalStates()

Get the array of final state (last BSM particle) particle objects in the element.

Returns list of Particle objects

hasTopInList(*elementList*)
Check if the element topology matches any of the topologies in the element list.

Parameters **elementList** – list of elements (Element objects)

Returns True, if element topology has a match in the list, False otherwise.

invisibleCompress()
Perform invisible compression.

Returns compressed copy of the element, if element ends with invisible particles; None, if compression is not possible

isRelatedTo(*other*)
Checks if the element has any common ancestors with other or one is an ancestor of the other. Returns True if self and other have at least one ancestor in common or are the same element, otherwise returns False.

Returns True/False

massCompress(*minmassgap*)
Perform mass compression.

Parameters **minmassgap** – value (in GeV) of the maximum mass difference for compression (if mass difference < minmassgap -> perform mass compression)

Returns compressed copy of the element, if two masses in this element are degenerate; None, if compression is not possible;

removeVertex(*ibr*, *iv*)
Remove vertex iv located in branch ibr. The "vertex-mother" in BSMparticles and (SM) particles in the vertex are removed from the branch. The vertex index corresponds to the BSM decay (iv = 0 will remove the first BSM particle,...)

Parameters

- **ibr** – Index of branch (int)
- **iv** – Index of vertex in branch ibr (int)

setCoveredBy(*resultType*)
Tag the element, all its daughter and all its mothers as covered by the type of result specified. It also recursively tags all granddaughters, grandmothers,...

Parameters **resultType** – String describing the type of result (e.g. 'prompt', 'displaced')

setEinfo()
Set topology info for each branch.

setTestedBy(*resultType*)
Tag the element, all its daughter and all its mothers as tested by the type of result specified. It also recursively tags all granddaughters, grandmothers,...

Parameters **resultType** – String describing the type of result (e.g. 'prompt', 'displaced')

sortBranches()
Sort branches. The smallest branch is the first one. See the Branch object for definition of branch size and comparison

switchBranches()
Switch branches, if the element contains a pair of them.

Returns element with switched branches (Element object)

toStr()
Returns a string with the element represented in bracket notation, including the final states, e.g. [[[jet]],[[jet]] (MET,MET)

theory.exceptions module

exception theory.exceptions.**SModelSTheoryError**(*value=None*)
Bases: Exception

Class to define SModelS specific errors

theory.lheReader module

class theory.lheReader.**LHEParticle**
Bases: object

An instance of this class represents a particle.

class theory.lheReader.**LheReader**(*filename*, *nmax=None*)
Bases: object

An instance of this class represents a reader for LHE files.

Constructor.

Parameters

- **filename** – LHE file name
- **nmax** – When using the iterator, then nmax is the maximum number of events to be reader, nmax=None means read till the end of the file. If filename is not a string, assume it is already a file object and do not open it.

close()
close file handle

event()
Get next event.

Returns SmsEvent; None if no event is left to be read.

next()
Get next element in iteration.

Needed for the iterator.

class theory.lheReader.**SmsEvent**(*eventnr=None*)
Bases: object

Event class featuring a list of particles and some convenience functions.

add(*particle*)
Add particle to the event.

getMom()
Return the pdgs of the mothers, None if a problem occurs.

metaInfo(*key*)
Return the meta information of 'key', None if info does not exist.

theory.lheReader.**getDictionariesFrom**(*lheFile*, *nevts=None*)

Reads all events in the LHE file and create mass and BR dictionaries for each branch in an event.

Parameters

- **lheFile** – LHE file
- **nevts** – (maximum) number of events used in the decomposition. If None, all events from file are processed.

Returns BR and mass dictionaries for the particles appearing in the event

theory.lheReader.**getDictionariesFromEvent**(*event*)

Create mass and BR dictionaries for each branch in an event.

Parameters **event** – LHE event

Returns BR and mass dictionaries for the branches in the event

theory.model module

class theory.model.**Model**(*BSMparticles*, *SMparticles*, *label=None*)

Bases: object

An instance of this class holds all the relevant information from the input model. This class contains the input file, the particles of the model and their production cross-sections.

Initializes the model :parameter BSMparticles: list with BSM particle objects :parameter SMparticles: list with SM particle objects :parameter label: Optional string to label the model

filterCrossSections()

Remove cross-sections for even particles or particles which do not belong to the model. Valid cross-sections are stored in self.xsections

Returns Number of cross-sections after filter.

getEvenOddList()

Get the list of even and odd particles, according to the Z2 parity value defined for each particle.

Returns list with PDGs of even particles, list with PDGs of odd particles

getModelDataFrom(*inputFile*)

Reads the input file (LHE or SLHA) and extract the relevant information (masses, widths, BRs and cross-sections). If a http address is given, it will attempt to download the file.

Parameters **inputFile** – input file (SLHA or LHE)

Returns dictionary with masses, dictionary with decays and XSectionList object

getParticlesWith(***kwargs*)

Return the particle objects with the listed attributes. MultiParticle objects are added if any of its particles matches the listed attributes. In order to avoid double counting, MultiParticle objects are only included if they do not contain any of the particles already in the list. For instance, if MP.particles = [e-,e+] and [e-] already appears in the list, MP will not be added.

Returns List of particle objects

getValuesFor(*attributeStr*)

Returns a list with all the values for attribute appearing in the model.

Parameters **attributeStr** – String for the desired attribute

Returns list of values

setDecays (*decaysDict*, *promptWidth*, *stableWidth*, *erasePrompt*)

setMasses (*massDict*, *roundMasses*)
Define particle masses using massDict.

Parameters

- **roundMasses** – If set, it will round the masses to this number of digits (int)
- **massDict** – dictionary with PDGs as keys and masses as values.

updateParticles (*inputFile, promptWidth=None, stableWidth=None, roundMasses=1, erasePrompt=['spin', 'eCharge', 'colordim']*)
Update mass, total width and branches of allParticles particles from input SLHA or LHE file.

Parameters

- **inputFile** – input file (SLHA or LHE)
- **promptWidth** – Maximum width for considering particles as decaying prompt. If None, it will be set 1e-8 GeV.
- **stableWidth** – Minimum width for considering particles as stable. If None, it will be set 1e-25 GeV.
- **roundMasses** – If set, it will round the masses to this number of digits (int)
- **erasePromptQNs** – If set, all particles with prompt decays (totalwidth > promptWidth) will have the corresponding properties (quantum numbers). So all particles with the same mass and Z2parity will be considered as equal when combining elements.

theory.particle module

class theory.particle.**MultiParticle**
Bases: *theory.particle.Particle*

An instance of this class represents a list of particle object to allow for inclusive expressions such as jets. The properties are: label, pdg, mass, electric charge, color charge, width

Creates a multiparticle. If a multiparticle with the exact same particles already been created return this multiparticle instead. Assigns an ID to the isntance using the class Particle._instance list. Reset the comparison dictionary.

Parameters

- **label** – Label for the MultiParticle (string)
- **particles** – List of Particle or MultiParticle objects (list)
- **attributesDict** – A dictionary with particle attributes (useful for pickling/unpickling). Attributes can also be directly assigned using keyword arguments.

cmpProperties (*other, properties=['Z2parity', 'spin', 'colordim', 'eCharge', 'mass', 'totalwidth']*)
Compares the properties in self with the ones in other. If other is a Particle object, checks if any of the particles in self matches other. If other is a MultiParticle object, checks if any particle in self matches any particle in other. If self and other are equal returns 0, else returns the result of comparing the first particle of self with other.

Parameters

- **other** – a Particle or MultiParticle object
- **properties** – list with properties to be compared. Default is spin, colordim and eCharge

Returns 0 if properties are equal, -1 if self < other and 1 if self > other.

contains(*particle*)
Check if MultiParticle contains the Particle object or MultiParticle object.

Parameters **particle** – Particle or MultiParticle object

Returns True/False

getLabels()
labels appearing in MultiParticle :return: list of labels of particles in the MultiParticle

getPdgs()
pdgs appearing in MultiParticle :return: list of pgds of particles in the MultiParticle

isMET()
Checks if all the particles in self can be considered as MET.

Returns True/False

isNeutral()
Return True if ALL particles in particle list are neutral.

Returns True/False

class theory.particle.**Particle**
Bases: object

An instance of this class represents a single particle. The properties are: label, pdg, mass, electric charge, color charge, width

Creates a particle. If a particle with the exact same attributes have already been created return this particle instead. Assigns an ID to the instance using the class Particle._instance list. Reset the comparison dictionary.

Parameters **attributesDict** – A dictionary with particle attributes (useful for pickling/unpickling). Attributes can also be directly assigned using keyword arguments.

Possible properties for arguments. Z2parity: int, +1 or -1 label: str, e.g. 'e-' pdg: number in pdg mass: mass of the particle echarge: electric charge as multiples of the unit charge colordim: color dimension of the particle spin: spin of the particle totalwidth: total width

chargeConjugate(*label=None*)
Returns the charge conjugate particle (flips the sign of eCharge). If it has a pdg property also flips its sign. If label is None, the charge conjugate name is defined as the original name plus "~" or if the original name ends in "+" ("-"), it is replaced by "-" ("+").

Parameters **label** – If defined, defines the label of the charge conjugated particle.

Returns the charge conjugate particle (Particle object)

cmpProperties(*other, properties=['Z2parity', 'spin', 'colordim', 'eCharge', 'mass', 'totalwidth']*)
Compare properties (default is spin, colordim and eCharge). Return 0 if properties are equal, -1 if self < other and 1 if self > other. Only compares the attributes which have been defined in both objects. The comparison is made in hierarchical order, following the order defined by the properties list.

Parameters

- **other** – a Particle or MultiParticle object
- **properties** – list with properties to be compared. Default is spin, colordim and eCharge

Returns 0 if properties are equal, -1 if self < other and 1 if self > other.

contains(*particle*)
If particle is a Particle object check if self and particle are the same object.

Parameters **particle** – Particle or MultiParticle object

Returns True/False

copy()
Make a copy of self with a distinct ID.

Returns A Particle object identical to self, except for its ID and comparison dict

describe()

eqProperties(*other, properties=['Z2parity', 'spin', 'colordim', 'eCharge', 'mass', 'totalwidth']*)
Check if particle has the same properties (default is spin, colordim and eCharge) as other. Only compares the attributes which have been defined in both objects.

Parameters

- **other** – a Particle or MultiParticle object
- **properties** – list with properties to be compared. Default is spin, colordim and eCharge

Returns True if all properties are the same, False otherwise.

classmethod getID()

classmethod getinstances()

isMET()
Checks if the particle can be considered as MET. If the _isInvisible attribute has not been defined, it will return True/False is isNeutral() = True/False. Else it will return the _isInvisible attribute.

Returns True/False

isNeutral()
Return True if the particle is electrically charged and color neutral. If these properties have not been defined, return True.

Returns True/False

isPrompt()
Checks if the particle decays promptly (e.g. totalwidth = inf).

Returns True/False

isStable()
Checks if the particle is stable (e.g. totalwidth = 0).

Returns True/False

class theory.particle.**ParticleList**
Bases: object

Simple class to store a list of particles.

Creates a particle list. If a list with the exact same particles have already been created return this list instead. Assigns an ID to the instance using the class ParticleList._instance list. Reset the comparison dictionary.

Parameters **particles** – List of Particle or MultiParticle objects (list)

classmethod getID()

classmethod getinstances()

theory.theoryPrediction module

theoryPrediction.**_getElementsFrom**(*smsTopList*, *dataset*)

Get elements that belong to any of the TxNames in dataset (appear in any of constraints in the result). Loop over all elements in smsTopList and returns a copy of the elements belonging to any of the constraints (i.e. have efficiency != 0). The copied elements have their weights multiplied by their respective efficiencies.

Parameters

- **dataset** – Data Set to be considered (DataSet object)
- **smsTopList** – list of topologies containing elements (TopologyList object)

Returns list of elements (Element objects)

class theory.theoryPrediction.**TheoryPrediction**

Bases: object

An instance of this class represents the results of the theory prediction for an analysis.

analysisId()

Return experimental analysis ID

computeStatistics(*marginalize=False*, *deltas_rel=0.2*)

Compute the likelihood, chi2 and expected upper limit for this theory prediction. The resulting values are stored as the likelihood and chi2 attributes. :param marginalize: if true, marginalize nuisances. Else, profile them. :param deltas_rel: relative uncertainty in signal (float). Default value is 20%.

dataId()

Return ID of dataset

dataType(*short=False*)

Return the type of dataset :param: short, if True, return abbreviation (ul,em,comb)

describe()

getLikelihood(*mu=1.0*, *marginalize=False*, *deltas_rel=0.2*, *expected=False*)

get the likelihood for a signal strength modifier mu :param expected: compute expected, not observed likelihood

getRValue(*expected=False*)

Get the r value = theory prediction / experimental upper limit

getUpperLimit(*expected=False*, *deltas_rel=0.2*)

Get the upper limit on sigma*eff. For UL-type results, use the UL map. For EM-Type returns the corresponding dataset (signal region) upper limit. For combined results, returns the upper limit on the total sigma*eff (for all signal regions/datasets).

Parameters

- **expected** – return expected Upper Limit, instead of observed.
- **deltas_rel** – relative uncertainty in signal (float). Default value is 20%.

Returns upper limit (Unum object)

getmaxCondition()

Returns the maximum xsection from the list conditions

Returns maximum condition xsection (float)

likelihoodFromLimits(*mu=1.0*, *marginalize=False*, *deltas_rel=0.2*, *expected=False*, *chi2also=False*)

compute the likelihood from expected and observed upper limits. :param expected: compute expected, not

observed likelihood :param mu: signal strength multiplier, applied to theory prediction :param chi2also: if true, return also chi2 :returns: likelihood; none if no expected upper limit is defined.

class theory.theoryPrediction.**TheoryPredictionList**(*theoryPredictions=None*, *maxCond=None*)

Bases: object

An instance of this class represents a collection of theory prediction objects.

Initializes the list.

Parameters

- **theoryPredictions** – list of TheoryPrediction objects
- **maxCond** – maximum relative violation of conditions for valid results. If defined, it will keep only the theory predictions with condition violation < maxCond.

append(*theoryPred*)

sortTheoryPredictions()

Reverse sort theoryPredictions by R value. Used for printer.

theory.theoryPrediction.**theoryPredictionsFor**(*expResult*, *smsTopList*, *maxMassDist=0.2*, *useBestDataset=True*, *combinedResults=True*, *marginalize=False*, *deltas_rel=0.2*)

Compute theory predictions for the given experimental result, using the list of elements in smsTopList. For each Txname appearing in expResult, it collects the elements and efficiencies, combine the masses (if needed) and compute the conditions (if exist).

Parameters

- **expResult** – expResult to be considered (ExpResult object)
- **smsTopList** – list of topologies containing elements (TopologyList object)
- **maxMassDist** – maximum mass distance for clustering elements (float)
- **useBestDataset** – If True, uses only the best dataset (signal region). If False, returns predictions for all datasets (if combinedResults is False), or only the combinedResults (if combinedResults is True).
- **combinedResults** – add theory predictions that result from combining datasets.
- **marginalize** – If true, marginalize nuisances. If false, profile them.
- **deltas_rel** – relative uncertainty in signal (float). Default value is 20%.

Returns a TheoryPredictionList object containing a list of TheoryPrediction objects

theory.topology module

class theory.topology.**Topology**(*elements=None*)

Bases: object

An instance of this class represents a topology.

Constructor. If elements is defined, create the topology from it. If elements it is a list, all elements must share a common global topology.

Parameters **elements** – Element object or list of Element objects

addElement(*newelement*)
Add an Element object to the elementList.

For all the pre-existing elements, which match the new element, add weight. If no pre-existing elements match the new one, add it to the list. OBS: newelement MUST ALREADY BE SORTED (see element.sort())

Parameters **newelement** – element to be added (Element object)

Returns True, if the element was added. False, otherwise

checkConsistency()
Check if the all the elements in elementList are consistent with the topology (same number of vertices and final states)

Returns True if all the elements are consistent. Print error message and exits otherwise.

describe()
Create a detailed description of the topology.

Returns list of strings with a description of the topology

getElements()
Get list of elements of the topology.

Returns elementList (list of Element objects)

getTotalWeight()
Return the sum of all elements weights.

Returns sum of weights of all elements (XSection object)

class theory.topology.**TopologyList**(*topologies=[]*)
Bases: object

An instance of this class represents an iterable collection of topologies.

Add topologies sequentially, if provided.

add(*newTopology*)
Check if elements in newTopology matches an entry in self.topos.

If it does, add weight. If the same topology exists, but not the same element, add element. If neither element nor topology exist, add the new topology and all its elements.

Parameters **newTopology** – Topology object

addElement(*newelement*)
Add an Element object to the corresponding topology in the list. If the element topology does not match any of the topologies in the list, create a new topology and insert it in the list. If the element topology already exists, add it to the respective topology. :parameter newelement: element to be added (Element object) :returns: True, if the element was added. False, otherwise

addList(*topoList*)
Adds topologies in topoList using the add method.

compressElements(*doCompress*, *doInvisible*, *minmassgap*)
Compress all elements in the list and include the compressed elements in the topology list.

Parameters

- **doCompress** – if True, perform mass compression
- **doInvisible** – if True, perform invisible compression

- **minmassgap** – value (in GeV) of the maximum mass difference for compression (if mass difference < minmassgap, perform mass compression)

describe()
Returns string with basic information about the topology list.

getElements()
Return a list with all the elements in all the topologies.

getTotalWeight()
Return the sum of all topologies total weights.

hasTopology(*topo*)
Checks if topo appears in any of the topologies in the list.

Parameters **topo** – Topology object

Returns True if topo appears in the list, False otherwise.

index(*topo*)
Uses bisect to find the index where of topo in the list. If topo does not appear in the list, returns None.

Parameters **topo** – Topology object

Returns position of topo in the list. If topo does not appear in the list, return None.

insert(*index*, *topo*)

Module contents

experiment package

Submodules

experiment.databaseObj module

class experiment.databaseObj.**Database**(*base=None*, *force_load=None*, *discard_zeroes=True*, *progressbar=False*, *subpickle=True*, *combinations-matrix=None*)

Bases: object

Database object. Holds a list of SubDatabases. Delegates all calls to SubDatabases.

Parameters

- **base** – path to the database, or pickle file (string), or http address. If None, "official", or "official_fastlim", use the official database for your code version (including fastlim results, if specified). If "latest", or "latest_fastlim", check for the latest database. Multiple databases may be specified using '+' as a delimiter.
- **force_load** – force loading the text database ("txt"), or binary database ("pcl"), dont force anything if None
- **discard_zeroes** – discard txnames with only zeroes as entries.
- **progressbar** – show a progressbar when building pickle file (needs the python-progressbar module)
- **subpickle** – produce small pickle files per exp result. Should only be used when working on the database.

- **combinationsmatrix** – an optional dictionary that contains info about combinable analyses, e.g. { “anaid1”: (“anaid2”, “anaid3”) } optionally specifying signal regions, e.g. { “anaid1:SR1”: (“anaid2:SR2”, “anaid3”) }

clearLinksToCombinationsMatrix()
clear all shallow links to the combinations matrix

createBinaryFile(*filename=None*)
create a pcl file from all the subs

createLinksToCombinationsMatrix()
in all globalInfo objects, create a shallow link to the combinations matrix

databaseParticles
Database particles, a list, one entry per sub

databaseVersion
The version of the database, concatenation of the individual versions

expResultList
The combined list, compiled from the individual lists

getExpResults(*analysisIDs=['all'], datasetIDs=['all'], txnames=['all'], dataTypes=['all'], useSuperseded=False, useNonValidated=False, onlyWithExpected=False*)
Returns a list of ExpResult objects.

Each object refers to an analysisID containing one (for UL) or more (for Efficiency maps) dataset (signal region) and each dataset containing one or more TxNames. If analysisIDs is defined, returns only the results matching one of the IDs in the list. If dataTypes is defined, returns only the results matching a dataType in the list. If datasetIDs is defined, returns only the results matching one of the IDs in the list. If txname is defined, returns only the results matching one of the Tx names in the list.

Parameters

- **analysisIDs** – list of analysis ids ([CMS-SUS-13-006,...]). Can be wildcarded with usual shell wildcards: * ? [<letters>] Furthermore, the centre-of-mass energy can be chosen as suffix, e.g. “:13*TeV”. Note that the asterisk in the suffix is not a wildcard.
- **datasetIDs** – list of dataset ids ([ANA-CUT0,...]). Can be wildcarded with usual shell wildcards: * ? [<letters>]
- **txnames** – list of txnames ([TChiWZ,...]). Can be wildcarded with usual shell wildcards: * ? [<letters>]
- **dataTypes** – dataType of the analysis (all, efficiencyMap or upperLimit) Can be wildcarded with usual shell wildcards: * ? [<letters>]
- **useSuperseded** – If False, the supersededBy results will not be included (deprecated)
- **useNonValidated** – If False, the results with validated = False will not be included
- **onlyWithExpected** – Return only those results that have expected values also. Note that this is trivially fulfilled for all efficiency maps.

Returns list of ExpResult objects or the ExpResult object if the list contains only one result

mergeERs(*o1*, *r2*)
merge the content of exp res r1 and r2

mergeLists(*lists*)
small function, merges lists of ERs

pcl_meta
The meta info of the text version, a merger of the original ones

txt_meta
The meta info of the text version, a merger of the original ones

class experiment.databaseObj.**ExpResultList**(*expResList*)
Bases: object

Holds a list of ExpResult objects for printout.

Parameters **expResultList** – list of ExpResult objects

class experiment.databaseObj.**SubDatabase**(*base=None*, *force_load=None*, *discard_zeroes=True*, *progressbar=False*, *subpickle=True*, *combinationsmatrix=None*)
Bases: object

SubDatabase object. Holds a list of ExpResult objects.

Parameters

- **base** – path to the database, or pickle file (string), or http address. If None, "official", or "official_fastlim", use the official database for your code version (including fastlim results, if specified). If "latest", or "latest_fastlim", check for the latest database. Multiple databases may be named, use "+" as delimiter. Order matters: Results with same name will overwritten according to sequence
- **force_load** – force loading the text database ("txt"), or binary database ("pcl"), dont force anything if None
- **discard_zeroes** – discard txnames with only zeroes as entries.
- **progressbar** – show a progressbar when building pickle file (needs the python-progressbar module)
- **subpickle** – produce small pickle files per exp result. Should only be used when working on the database.
- **combinationsmatrix** – an optional dictionary that contains info about combinable analyses, e.g. { "anaid1": ("anaid2", "anaid3") } optionally specifying signal regions, e.g. { "anaid1:SR1": ("anaid2:SR2", "anaid3") }

base
This is the path to the base directory.

checkBinaryFile()

checkPathName(*path*, *discard_zeroes*)
checks the path name, returns the base directory and the pickle file name. If path starts with http or ftp, fetch the description file and the database. returns the base directory and the pickle file name

clearLinksToCombinationsMatrix()

createBinaryFile(*filename=None*)
create a pcl file from the text database, potentially overwriting an old pcl file.

createExpResult(*root*)
create, from pickle file or text files

createLinksToCombinationsMatrix()
in all globalInfo objects, create links to self.combinationsmatrix

createLinksToModel()
in all globalInfo objects, create links to self.databaseParticles

databaseVersion
The version of the database, read from the 'version' file.

fetchFromScratch(*path*, *store*, *discard_zeroes*)
fetch database from scratch, together with description. :param store: filename to store json file.

fetchFromServer(*path*, *discard_zeroes*)

getExpResults(*analysisIDs=['all'], datasetIDs=['all'], txnames=['all'], dataTypes=['all'], useSuperseded=False, useNonValidated=False, onlyWithExpected=False*)
Returns a list of ExpResult objects.

Each object refers to an analysisID containing one (for UL) or more (for Efficiency maps) dataset (signal region) and each dataset containing one or more TxNames. If analysisIDs is defined, returns only the results matching one of the IDs in the list. If dataTypes is defined, returns only the results matching a dataType in the list. If datasetIDs is defined, returns only the results matching one of the IDs in the list. If txname is defined, returns only the results matching one of the Tx names in the list.

Parameters

- **analysisIDs** – list of analysis ids ([CMS-SUS-13-006,...]). Can be wildcarded with usual shell wildcards: * ? [<letters>] Furthermore, the centre-of-mass energy can be chosen as suffix, e.g. ":13*TeV". Note that the asterisk in the suffix is not a wildcard.
- **datasetIDs** – list of dataset ids ([ANA-CUT0,...]). Can be wildcarded with usual shell wildcards: * ? [<letters>]
- **txnames** – list of txnames ([TChiWZ,...]). Can be wildcarded with usual shell wildcards: * ? [<letters>]
- **dataTypes** – dataType of the analysis (all, efficiencyMap or upperLimit) Can be wildcarded with usual shell wildcards: * ? [<letters>]
- **useSuperseded** – If False, the supersededBy results will not be included (deprecated)
- **useNonValidated** – If False, the results with validated = False will not be included
- **onlyWithExpected** – Return only those results that have expected values also. Note that this is trivially fulfilled for all efficiency maps.

Returns list of ExpResult objects or the ExpResult object if the list contains only one result

inNotebook()
Are we running within a notebook? Has an effect on the progressbar we wish to use.

loadBinaryFile(*lastm_only=False*)
Load a binary database, returning last modified, file count, database.

Parameters **lastm_only** – if true, the database itself is not read.

Returns database object, or None, if lastm_only == True.

loadDatabase()
if no binary file is available, then load the database and create the binary file. if binary file is available, then check if it needs update, create new binary file, in case it does need an update.

loadTextDatabase()
simply loads the textdabase

needsUpdate()
does the binary db file need an update?

removeLinksToModel()
remove the links of globalInfo._databaseParticles to the model. Currently not used.

updateBinaryFile()
write a binar db file, but only if necessary.

experiment.datasetObj module

class experiment.datasetObj.**CombinedDataSet**(*expResult*)
Bases: object

Holds the information for a combined dataset (used for combining multiple datasets).

combinedLikelihood(*nsig*, *marginalize=False*, *deltas_rel=0.2*)
Computes the (combined) likelihood to observe nobs events, given a predicted signal "nsig", with nsig being a vector with one entry per dataset. nsig has to obey the datasetOrder. Deltas is the error on the signal. :param nsig: predicted signal (list) :param deltas_rel: relative uncertainty in signal (float). Default value is 20%.

Returns likelihood to observe nobs events (float)

getCombinedUpperLimitFor(*nsig*, *expected=False*, *deltas_rel=0.2*)
Get combined upper limit. If covariances are given in globalInfo then simplified likelihood is used, else if json files are given pyhf cimbination is performed.

Parameters

- **nsig** – list of signal events in each signal region/dataset. The list should obey the ordering in globalInfo.datasetOrder.
- **expected** – return expected, not observed value
- **deltas_rel** – relative uncertainty in signal (float). Default value is 20%.

Returns upper limit on sigma*eff

getDataSet(*datasetID*)
Returns the dataset with the corresponding dataset ID. If the dataset is not found, returns None.

Parameters **datasetID** – dataset ID (string)

Returns DataSet object if found, otherwise None.

getID()
Return the ID for the combined dataset

getLumi()
Return the dataset luminosity. For CombinedDataSet always return the value defined in globalInfo.lumi.

getPyhfComputer(*nsig*)
create the pyhf ul computer object :returns: pyhf upper limit computer, and combinations of signal regions

getType()
Return the dataset type (combined)

sortDataSets()
Sort datasets according to globalInfo.datasetOrder.

totalChi2(*nsig*, *marginalize=False*, *deltas_rel=0.2*)
Computes the total chi2 for a given number of observed events, given a predicted signal "nsig", with nsig being a vector with one entry per dataset. nsig has to obey the datasetOrder. Deltas is the error on the signal efficiency. :param nsig: predicted signal (list) :param deltas_rel: relative uncertainty in signal (float). Default value is 20%.

Returns chi2 (float)

class experiment.datasetObj.**DataSet**(*path=None*, *info=None*, *createInfo=True*, *discard_zeroes=True*, *databaseParticles=None*)
Bases: object

Holds the information to a data set folder (TxName objects, dataInfo,...)

Parameters **discard_zeroes** – discard txnames with zero-only results

checkForRedundancy(*databaseParticles*)
In case of efficiency maps, check if any txnames have overlapping constraints. This would result in double counting, so we dont allow it.

chi2(*nsig*, *deltas_rel=0.2*, *marginalize=False*)
Computes the chi2 for a given number of observed events "nobs", given number of signal events "nsig", and error on signal "deltas". nobs, expectedBG and bgError are part of dataInfo. :param nsig: predicted signal (float) :param deltas_rel: relative uncertainty in signal (float). Default value is 20%. :param marginalize: if true, marginalize nuisances. Else, profile them. :return: chi2 (float)

folderName()
Name of the folder in text database.

getAttributes(*showPrivate=False*)
Checks for all the fields/attributes it contains as well as the attributes of its objects if they belong to smodels.experiment.

Parameters **showPrivate** – if True, also returns the protected fields (_field)

Returns list of field names (strings)

getEfficiencyFor(*txname*, *mass*)
Convenience function. Get efficiency for mass assuming no lifetime rescaling. Same as self.getTxName(txname).getEfficiencyFor(m)

getID()
Return the dataset ID

getLumi()
Return the dataset luminosity. If not defined for the dataset, use the value defined in globalInfo.lumi.

getSRUpperLimit(*alpha=0.05*, *expected=False*, *compute=False*, *deltas_rel=0.2*)
Computes the 95% upper limit on the signal*efficiency for a given dataset (signal region). Only to be used for efficiency map type results.

Parameters

- **alpha** – Can be used to change the C.L. value. The default value is 0.05 (= 95% C.L.)
- **expected** – Compute expected limit (i.e. Nobserved = NexpectedBG)
- **deltas_rel** – relative uncertainty in signal (float). Default value is 20%.
- **compute** – If True, the upper limit will be computed from expected and observed number of events. If False, the value listed in the database will be used instead.

Returns upper limit value

getTxName(*txname*)
get one specific txName object.

getType()
Return the dataset type (EM/UL)

getUpperLimitFor(*element=None*, *expected=False*, *txnames=None*, *compute=False*, *alpha=0.05*, *deltas_rel=0.2*)
Returns the upper limit for a given element (or mass) and txname. If the dataset hold an EM map result the upper limit is independent of the input txname or mass. For UL results if an Element object is given the corresponding upper limit will be rescaled according to the lifetimes of the element intermediate particles. On the other hand, if a mass is given, no rescaling will be applied.

Parameters

- **txname** – TxName object or txname string (only for UL-type results)
- **element** – Element object or mass array with units (only for UL-type results)
- **alpha** – Can be used to change the C.L. value. The default value is 0.05 (= 95% C.L.) (only for efficiency-map results)
- **deltas_rel** – relative uncertainty in signal (float). Default value is 20%.
- **expected** – Compute expected limit, i.e. Nobserved = NexpectedBG (only for efficiency-map results)
- **compute** – If True, the upper limit will be computed from expected and observed number of events. If False, the value listed in the database will be used instead.

Returns upper limit (Unum object)

getValuesFor(*attribute*)

Returns a list for the possible values appearing in the ExpResult for the required attribute (sqrts,id,constraint,...). If there is a single value, returns the value itself.

Parameters **attribute** – name of a field in the database (string).

Returns list of unique values for the attribute

isCombMatrixCombinableWith_(*other*)

check for combinability via the combinations matrix

isCombinableWith(*other*)

Function that reports if two datasets are mutually uncorrelated = combinable.

Parameters **other** – datasetObj to compare self with

isGlobalFieldCombinableWith_(*other*)

check for 'combinableWith' fields in globalInfo, check if <other> matches. this check is at analysis level (not at dataset level).

Params other a dataset to check against

Returns true, if pair is marked as combinable, else false

isLocalFieldCombinableWith_(*other*)

check for 'combinableWith' fields in globalInfo, check if <other> matches. this check is at dataset level (not at dataset level).

Params other a dataset to check against

Returns true, if pair is marked as combinable, else false

likelihood(*nsig*, *deltas_rel=0.2*, *marginalize=False*, *expected=False*)

Computes the likelihood to observe nobs events, given a predicted signal "nsig", assuming "deltas" error on the signal efficiency. The values observedN, expectedBG, and bgError are part of dataInfo.

Parameters

- **nsig** – predicted signal (float)
- **deltas_rel** – relative uncertainty in signal (float). Default value is 20%.
- **marginalize** – if true, marginalize nuisances. Else, profile them.
- **expected** – Compute expected instead of observed likelihood

Returns likelihood to observe nobs events (float)

experiment.defaultFinalStates module

experiment.exceptions module

exception experiment.exceptions.**DatabaseNotFoundException**(*value*)
Bases: Exception

This exception is used when the database cannot be found.

exception experiment.exceptions.**SModelSExperimentError**(*value=None*)
Bases: Exception

Class to define SModelS specific errors

experiment.expResultObj module

class experiment.expResultObj.**ExpResult**(*path=None*, *discard_zeroes=True*, *databaseParticles=None*)
Bases: object

Object containing the information and data corresponding to an experimental result (experimental conference note or publication).

Parameters

- **path** – Path to the experimental result folder
- **discard_zeroes** – Discard maps with only zeroes
- **databaseParticles** – the model, i.e. the particle content

getAttributes(*showPrivate=False*)
Checks for all the fields/attributes it contains as well as the attributes of its objects if they belong to smodels.experiment.

Parameters **showPrivate** – if True, also returns the protected fields (_field)

Returns list of field names (strings)

getDataset(*dataId*)
retrieve dataset by dataId

getEfficiencyFor(*txname*, *mass*, *dataset=None*)
Convenience function. Get the efficiency for a specific dataset for a a specific txname. Equivalent to: self.getDataset (dataset).getEfficiencyFor (txname, mass)

getTxNames()
Returns a list of all TxName objects appearing in all datasets.

getTxnameWith(*restrDict={}*)
Returns a list of TxName objects satisfying the restrictions. The restrictions specified as a dictionary.

Parameters **restrDict** – dictionary containing the fields and their allowed values. E.g. {'txname' : 'T1', 'axes' : ....} The dictionary values can be single entries or a list of values. For the fields not listed, all values are assumed to be allowed.

Returns list of TxName objects if more than one txname matches the selection criteria or a single TxName object, if only one matches the selection.

getUpperLimitFor(*dataID=None*, *alpha=0.05*, *expected=False*, *txname=None*, *mass=None*, *compute=False*)
Computes the 95% upper limit (UL) on the signal cross section according to the type of result. For an

Efficiency Map type, returns the UL for the signal*efficiency for the given dataSet ID (signal region). For an Upper Limit type, returns the UL for the signal*BR for the given mass array and Txname.

Parameters

- **dataID** – dataset ID (string) (only for efficiency-map type results)
- **alpha** – Can be used to change the C.L. value. The default value is 0.05 (= 95% C.L.) (only for efficiency-map results)
- **expected** – Compute expected limit, i.e. Nobserved = NexpectedBG (only for efficiency-map results)
- **txname** – TxName object or txname string (only for UL-type results)
- **mass** – Mass array with units (only for UL-type results)
- **compute** – If True, the upper limit will be computed from expected and observed number of events. If False, the value listed in the database will be used instead.

Returns upper limit (Unum object)

getValuesFor(*attribute*)
Returns a list for the possible values appearing in the ExpResult for the required attribute (sqrts,id,constraint,...). If there is a single value, returns the value itself.

Parameters **attribute** – name of a field in the database (string).

Returns list of unique values for the attribute

hasCovarianceMatrix()

hasJsonFile()

id()

writePickle(*dbVersion*)
write the pickle file

experiment.infoObj module

class experiment.infoObj.**Info**(*path=None*)
Bases: object

Holds the meta data information contained in a .txt file (luminosity, sqrts, experimentID,...). Its attributes are generated according to the lines in the .txt file which contain "info_tag: value".

Parameters **path** – path to the .txt file

addInfo(*tag*, *value*)
Adds the info field labeled by tag with value value to the object.

Parameters

- **tag** – information label (string)
- **value** – value for the field in string format

cacheJsons()
if we have the "jsonFiles" attribute defined, we cache the corresponding jsons. Needed when pickling

dirName(*up=0*)
directory name of path. If up>0, we step up 'up' directory levels.

getInfo(*infoLabel*)
Returns the value of info field.

Parameters **infoLabel** – label of the info field (string). It must be an attribute of the GlobalInfo object

experiment.metaObj module

class experiment.metaObj.**Meta**(*pathname*, *discard_zeroes=None*, *mtime=None*, *filecount=None*, *hasFastLim=None*, *databaseVersion=None*, *format_version=214*, *python='3.6.12 (default*, *Oct 19 2020*, *15:18:45) n[GCC 7.5.0]'*)

Bases: object

Parameters

- **pathname** – filename of pickle file, or dirname of text files
- **discard_zeroes** – do we discard zeroes?
- **mtime** – last modification time stamps
- **filecount** – number of files
- **hasFastLim** – fastlim in the database?
- **databaseVersion** – version of database
- **format_version** – format version of pickle file
- **python** – python version

cTime()

current_version = 214
The Meta object holds all meta information regarding the database, like number of analyses, last time of modification, ... This info is needed to understand if we have to re-pickle.

determineLastModified(*force=False*)
compute the last modified timestamp, plus count number of files. Only if text db

getPickleFileName()
get canonical pickle file name

isPickle()
is this meta info from a pickle file?

lastModifiedSubDir(*subdir*)
Return the last modified timestamp of subdir (working recursively) plus the number of files.

Parameters

- **subdir** – directory name that is checked
- **lastm** – the most recent timestamp so far, plus number of files

Returns the most recent timestamp, and the number of files

needsUpdate(*current*)
do we need an update, with respect to <current>. so <current> is the text database, <self> the pcl.

printFastlimBanner()
check if fastlim appears in data. If yes, print a statement to stdout.

sameAs(*other*)
check if it is the same database version

versionFromFile()
Retrieves the version of the database using the version file.

experiment.txnameObj module

class experiment.txnameObj.**Delaunay1D**(*data*)
Bases: object

Uses a 1D data array to interpolate the data. The attribute simplices is a list of N-1 pair of ints with the indices of the points forming the simplices (e.g. [[0,1],[1,2],[3,4],...]).

checkData(*data*)
Define the simplices according to data. Compute and store the transformation matrix and simplices self.point.

find_index(*xlist*, *x*)
Efficient way to find x in a list. Returns the index (i) of xlist such that xlist[i] < x <= xlist[i+1]. If x > max(xlist), returns the length of the list. If x < min(xlist), returns 0. vertices = np.take(self.tri.simplices, simplex, axis=0) temp = np.take(self.tri.transform, simplex, axis=0) d=temp.shape[2] delta = uvw - temp[:, d]

Parameters

- **xlist** – List of x-type objects
- **x** – object to be searched for.

Returns Index of the list such that xlist[i] < x <= xlist[i+1].

find_simplex(*x*, *tol=0.0*)
Find 1D data interval (simplex) to which x belongs

Parameters

- **x** – Point (float) without units
- **tol** – Tolerance. If x is outside the data range with distance < tol, extrapolate.

Returns simplex index (int)

class experiment.txnameObj.**TxName**(*path*, *globalObj*, *infoObj*, *databaseParticles*)
Bases: object

Holds the information related to one txname in the Txname.txt file (constraint, condition,...) as well as the data.

addInfo(*tag*, *value*)
Adds the info field labeled by tag with value value to the object.

Parameters

- **tag** – information label (string)
- **value** – value for the field in string format

fetchAttribute(*attr*, *fillvalue=None*)
Auxiliary method to get the attribute from self. If not found, look for it in datasetInfo and if still not found look for it in globalInfo. If not found in either of the above, return fillvalue.

Parameters

- **attr** – Name of attribute (string)
- **fillvalue** – Value to be returned if attribute is not found.

Returns Value of the attribute or fillvalue, if attribute was not found.

getEfficiencyFor(*element*)
For upper limit results, checks if the input element falls inside the upper limit grid and has a non-zero reweigthing factor. If it does, returns efficiency = 1, else returns efficiency = 0. For efficiency map results, returns the signal efficiency including the lifetime reweighting. If a mass array is given as input, no lifetime reweighting will be applied.

Parameters **element** – Element object or mass array with units.

Returns efficiency (float)

getInfo(*infoLabel*)
Returns the value of info field.

Parameters **infoLabel** – label of the info field (string). It must be an attribute of the TxNameInfo object

getMassVectorFromElement(*element*)
given element, extract the mass vector for the server query. element can be list of masses or "Element"

Returns eg [[300,100],[300,100]]

getQueryStringForElement(*element*)

getULFor(*element*, *expected=False*)
Returns the upper limit (or expected) for element (only for upperLimit-type). Includes the lifetime reweighting (ul/reweight). If called for efficiencyMap results raises an error. If a mass array is given as input, no lifetime reweighting will be applied.

Parameters

- **element** – Element object or mass array (with units)
- **expected** – look in self.txnameDataExp, not self.txnameData

hasElementAs(*element*)
Verify if the conditions or constraint in Txname contains the element. Check both branch orderings. If both orderings match, returns the one with the highest mass array.

Parameters **element** – Element object

Returns A copy of the element on the correct branch ordering appearing in the Txname constraint or condition.

hasLikelihood()
can I construct a likelihood for this map? True for all efficiency maps, and for upper limits maps with expected Values.

hasOnlyZeroes()

class experiment.txnameObj.**TxNameData**(*value*, *dataType*, *Id*, *accept_errors_upto=0.05*, *Leff_inner=None*, *Leff_outer=None*)
Bases: object

Holds the data for the Txname object. It holds Upper limit values or efficiencies.

Parameters

- **value** – values in string format
- **dataType** – the dataType (upperLimit or efficiencyMap)
- **Id** – an identifier, must be unique for each TxNameData!

- **_accept_errors_upto** – If None, do not allow extrapolations outside of convex hull. If float value given, allow that much relative uncertainty on the upper limit / efficiency when extrapolating outside convex hull. This method can be used to loosen the equal branches assumption.
- **Leff_inner** – is the effective inner radius of the detector, given in meters (used for reweighting prompt decays). If None, default values will be used.
- **Leff_outer** – is the effective outer radius of the detector, given in meters (used for reweighting decays outside the detector). If None, default values will be used.

computeV(*values*)

Compute rotation matrix _V, and triangulation self.tri

Parameters **values** – Nested array with the data values without units

coordinatesToData(*point*, *rotMatrix=None*, *transVector=None*)

A function that return the original mass and width array (including the widths as tuples) for a given point in PCA space (inverse of dataToCoordinates).

Parameters

- **point** – Point in PCA space (1D list with size equal to self.full_dimensionality or self.dimensionality)
- **rotMatrix** – Rotation matrix for PCA (e.g. self._V). If None, no rotation is performed.
- **transVector** – Translation vector for PCA (e.g. self.delta_x). If None no translation is performed

Returns nested mass array including the widths as tuples (e.g. [[(200,1e-10),100],[(200,1e-10),100]])

countNonZeros(*mp*)

count the nonzeros in a vector

dataToCoordinates(*dataPoint*, *rotMatrix=None*, *transVector=None*)

Format a dataPoint to the format used for interpolation. All the units are removed, the widths are rescaled and the masses and widths are combined in a flat array. The input can be an Element object or a massAndWidth nested arrays (with tuples to store the relevant widths).

Parameters

- **dataPoint** – Element object from which the mass and width arrays will be extracted or a nested mass array from the database, which contain tuples to include the width values
- **rotMatrix** – Rotation matrix for PCA (e.g. self._V). If None, no rotation is performed.
- **transVector** – Translation vector for PCA (e.g. self.delta_x). If None no translation is performed

Returns Point (list of floats)

evaluateString(*value*)

Evaluate string.

Parameters **value** – String expression.

getDataShape(*value*)

Stores the data format (mass shape) and store it for future use. If there are inclusive objects (mass or branch = None), store their positions.

Parameters **value** – list of data points

getUnits(*value*)
Get standard units for the input object. Uses the units defined in physicsUnits.standardUnits. (e.g. [[100*GeV,100.*GeV],3.*pb] -> returns [[GeV,GeV],fb] [[100*GeV,3.],[200.*GeV,2.*pb]] -> returns [[GeV,1.],[GeV,fb]])

Parameters **value** – Object containing units (e.g. [[100*GeV,100.*GeV],3.*pb])

Returns Object with same structure containing the standard units used to normalize the data.

getValueFor(*element*)
Interpolates the value and returns the UL or efficiency for the respective element rescaled according to the reweighting function self.reweightF. For UL-type data the default rescaling is ul -> ul/(fraction of prompt decays) and for EM-type data it is eff -> eff*(fraction of prompt decays). If a mass array is given as input, no lifetime reweighting will be applied.

Parameters **element** – Element object or mass array (with units)

getValueForPoint(*point*)
Returns the UL or efficiency for the point (in coordinates) using interpolation

Parameters **point** – Point in coordinate space (length = self.full_dimensionality)

Returns Value of UL or efficiency (float) without units

getWidthPosition(*value*)
Gets the positions of the widths to be used for interpolation.

Parameters **value** – data point

Returns A list with the position of the widths. A position is a tuple of the form (branch-index,vertex-index).

interpolate(*point*, *fill_value=nan*)
Returns the interpolated value for the point (in coordinates)

Parameters **point** – Point in coordinate space (length = self.dimensionality)

Returns Value for point without units

loadData(*value*)
Uses the information in value to generate the data grid used for interpolation.

onlyZeroValues()
check if the map is zeroes only

round_to_n(*x*, *n*)

Module contents

tools package

Submodules

tools.asciiGraph module

tools.asciiGraph.**asciidraw**(*element*, *labels=True*, *html=False*, *border=False*)
Draw a simple ASCII graph on the screen.

tools.caching module

class tools.caching.**Cache**
Bases: object

a class for storing results from interpolation

static add(*key*, *value*)

n_stored = 1000

static reset()
completely reset the cache

static size()

tools.colors module

class tools.colors.**Colors**
Bases: object

blue

cyan

debug

error

green

info

magenta

red

reset

warn

yellow

tools.coverage module

class tools.coverage.**GeneralElement**(*el*, *missingX*, *smFinalStates*, *bsmFinalStates*)
Bases: object

This class represents a simplified (general) element which does only holds information about its even particles and decay type. The even particles are replaced/grouped by the particles defined in smFinalStates.

sortBranches()
Sort branches. The smallest branch is the first one. If branches are equal, sort accoding to decayType.

class tools.coverage.**Uncovered**(*topoList, sqrts=None, sigmacut=0.00E+00 [fb], groupFilters={'missing (all)': <function <lambda>>, 'missing (displaced)': <function <lambda>>, 'missing (prompt)': <function <lambda>>, 'outsideGrid (all)': <function <lambda>>}, groupFactors={'missing (all)': <function <lambda>>, 'missing (displaced)': <function <lambda>>, 'missing (prompt)': <function <lambda>>, 'outsideGrid (all)': <function <lambda>>}, groupdDescriptions={'missing (all)': 'missing topologies', 'missing (displaced)': 'missing topologies with displaced decays', 'missing (prompt)': 'missing topologies with prompt decays', 'outsideGrid (all)': 'topologies outside the grid'}, smFinalStates=None, bsmFinalSates=None*)

Bases: object

Wrapper object for defining and holding a list of coverage groups (UncoveredGroup objects).

The class builds a series of UncoveredGroup objects and stores them.

Inititalize the object.

Parameters

- **topoList** – TopologyList object used to select elements from.
- **sqrts** – Value (with units) for the center of mass energy used to compute the missing cross sections. If not specified the largest value available will be used.
- **sigmacut** – Minimum cross-section/weight value (after applying the reweight factor) for an element to be included. The value should in fb (unitless)
- **groupFilters** – Dictionary containing the groups' labels and the method for selecting elements.
- **groupFactors** – Dictionary containing the groups' labels and the method for reweighting cross sections.
- **groupdDescriptions** – Dictionary containing the groups' labels and strings describing the group (used for printout)
- **smFinalStates** – List of (inclusive) Particle or MultiParticle objects used for grouping Z2-even particles when creating GeneralElements.
- **bsmFinalSates** – List of (inclusive) Particle or MultiParticle objects used for grouping Z2-odd particles when creating GeneralElements.

getGroup(*label*)

Returns the group with the required label. If not found, returns None.

Parameters **label** – String corresponding to the specific group label

Returns UncoveredGroup object which matches the label

class tools.coverage.**UncoveredGroup**(*label*, *elementFilter*, *reweightFactor*, *smFinalStates*, *bsmFinalStates*, *sqrts*, *sigmacut=0.0*)

Bases: object

Holds information about a single coverage group: criteria for selecting and grouping elements, function for reweighting cross sections, etc.

Parameters

- **label** – Group label

- **elementFilter** – Function which takes an element as argument and returns True (False) if the element should (not) be selected.
- **reweightFactor** – Function which takes an element as argument and returns the reweighting factor to be applied to the element weight.
- **smFinalStates** – List of Particle/MultiParticle objects used to group Z2-even particles appearing in the final state
- **bsmFinalStates** – List of Particle/MultiParticle objects used to group Z2-odd particles appearing in the final state
- **sqrts** – Value (with units) for the center of mass energy used to compute the missing cross sections. If not specified the largest value available will be used.
- **sigmacut** – Minimum cross-section/weight value (after applying the reweight factor) for an element to be included. The value should in fb (unitless)

addToGeneralElements(*el*, *missingX*)
Adds an element to the list of missing topologies = general elements. If the element contributes to a missing topology that is already in the list, add element and weight to topology. :parameter el: element to be added :parameter missingX: missing cross-section for the element (in fb)

getMissingX(*element*)
Calculate total missing cross section of an element, by recursively checking if its mothers already appear in the list. :param element: Element object

Returns missing cross section without units (in fb)

getToposFrom(*topoList*)
Select the elements from topoList according to self.elementFilter and build GeneralElements from the selected elements. The GeneralElement weights corresponds to the missing cross-section with double counting from compressed elements already accounted for.

getTotalXSec(*sqrts=None*)
Calculate total missing topology cross section at sqrts. If no sqrts is given use self.sqrts :ivar sqrts: sqrts

tools.crashReport module

class tools.crashReport.**CrashReport**
Bases: object

Class that handles all crash report information.

createCrashReportFile(*inputFileName*, *parameterFileName*)
Create a new SModelS crash report file.

A SModelS crash report file contains:

- a timestamp
- SModelS version
- platform information (CPU architecture, operating system, ...)
- Python version
- stack trace
- input file name
- input file content

- parameter file name
- parameter file content

Parameters

- **inputFileName** – relative location of the input file
- **parameterFileName** – relative location of the parameter file

createUnknownErrorMessage()
Create a message for an unknown error.

tools.crashReport.**createStackTrace**()
Return the stack trace.

tools.crashReport.**readCrashReportFile**(*crashReportFileName*)
Read a crash report file to use its input and parameter file sections for a SModelS run.

Parameters **crashReportFileName** – relative location of the crash report file

tools.databaseBrowser module

class tools.databaseBrowser.**Browser**(*database*, *force_txt=False*)
Bases: object

Browses the database, exits if given path does not point to a valid smodels-database. Browser can be restricted to specified run or experiment.

Parameters

- **force_txt** – If True forces loading the text database.
- **database** – Path to the database or Database object

getAttributes(*showPrivate=False*)
Checks for all the fields/attributes it contains as well as the attributes of its objects if they belong to smodels.experiment.

Parameters **showPrivate** – if True, also returns the protected fields (_field)

Returns list of field names (strings)

getEfficiencyFor(*expid*, *dataset*, *txname*, *massarray*)
Get an efficiency for the given experimental id, the dataset name, the txname, and the massarray. Can only be used for EfficiencyMap-type experimental results. Interpolation is done, if necessary.

Parameters

- **expid** – experimental id (string)
- **dataset** – dataset name (string)
- **txname** – txname (string).
- **massarray** – list of masses with units, e.g. [[400.*GeV, 100.*GeV],[400.*GeV, 100.*GeV]]

Returns efficiency

getULFor(*expid*, *txname*, *massarray*, *expected=False*)
Get an upper limit for the given experimental id, the txname, and the massarray. Can only be used for UL experimental results. Interpolation is done, if necessary.

Parameters

- **expid** – experimental id (string)
- **txname** – txname (string). ONLY required for upper limit results
- **massarray** – list of masses with units, e.g. [[400.*GeV, 100.*GeV],[400.*GeV, 100.*GeV]]
- **expected** – If true, return expected upper limit, otherwise return observed upper limit.

Returns upper limit [fb]

getULForSR(*expid*, *datasetID*)
Get an upper limit for the given experimental id and dataset (signal region). Can only be used for efficiency-map results. :param expid: experimental id (string) :param datasetID: string defining the dataset id, e.g. ANA5-CUT3. :return: upper limit [fb]

getValuesFor(*attribute*, *expResult=None*)
Returns a list for the possible values appearing in the database for the required attribute (sqrts,id,constraint,...).

Parameters

- **attribute** – name of a field in the database (string).
- **expResult** – if defined, restricts the list to the corresponding expResult. Must be an ExpResult object.

Returns list of values

loadAllResults()
Saves all the results from database to the _selectedExpResults. Can be used to restore all results to _selectedExpResults.

selectExpResultsWith(***restrDict*)
Loads the list of the experimental results (pair of InfoFile and DataFile) satisfying the restrictions to the _selectedExpResults. The restrictions specified as a dictionary.

Parameters **restrDict** – selection fields and their allowed values. E.g. lumi = [19.4/fb, 20.3/fb], txName = 'T1',....} The values can be single entries or a list of values. For the fields not listed, all values are assumed to be allowed.

tools.databaseBrowser.**main**(*args*)
IPython interface for browsing the Database.

tools.databaseClient module

class tools.databaseClient.**DatabaseClient**(*servername=None*, *port=None*, *verbose='info'*, *rundir='./'*, *logfile='@@rundir@@/dbclient.log'*, *clientid=-1*)

Bases: object

clearCache()

findServerStatus()

getWaitingTime()
compute a waiting time between attempts, from self.ntries

initialize()

log(**args*)

nameAndPort()

pprint(**args*)

query(*msg*)
query a certain result, msg is eg. obs:ATLAS-SUSY-2016-07:ul:T1:[[5.5000E+02,4.5000E+02],[5.5000E+02,4.5000E+02]]

saveStats()

send(*message*, *amount_expected=32*)
send the message. :param amount_expected: how many return bytes do you expect

send_shutdown()
send shutdown request to server

setDefaults()
put in some defaults if data members dont exist

tools.databaseClient.**stresstest**(*args*)
this is one process in the stress test

tools.databaseServer module

class tools.databaseServer.**DatabaseServer**(*dbpath*, *servername=None*, *port=None*, *verbose='info'*, *rundir='./'*, *logfile='@@rundir@@/dbserver.log'*)

Bases: object

finish()

initialize()

is_port_in_use(*port*)
check if port is in use

listen()

log(**args*)

logServerStats()
log our stats upon exit

lookUpResult(*data*)

parseData(*data*)
parse the data packet

pprint(**args*)

run(*nonblocking=False*)
run server :param nonblock: run in nonblocking mode (not yet implemented)

setStatus(*status*)
servers have a status file that tells us if they are running

shutdown(*fromwhere='unknown'*)

tools.databaseServer.**shutdownAll**()

tools.externalPythonTools module

class tools.externalPythonTools.**ExternalPythonTool**(*importname*, *optional=False*)
Bases: object

An instance of this class represents the installation of a python package. As it is python-only, we need this only for installation, not for running (contrary to nllfast or pythia).

Initializes the ExternalPythonTool object. Useful for installation. :params optional: optional package, not needed for core SModelS.

checkInstallation()
The check is basically done in the constructor

compile()

installDirectory()
Just returns the pythonPath variable

pathOfExecutable()
Just returns the pythonPath variable

tools.inclusiveObjects module

class tools.inclusiveObjects.**InclusiveList**
Bases: list

An inclusive list class. It will return True when compared to any other list object.

class tools.inclusiveObjects.**InclusiveValue**
Bases: int

An inclusive number class. It will return True when compared to any other integer, float or Unum object.

tools.interactivePlots module

class tools.interactivePlots.**Plotter**(*smodelsFolder*, *slhaFolder*, *parameterFile*, *modelFile=None*)
Bases: object

A class to store the required data and produce the interactive plots

Initializes the class.

Parameters

- **smodelsFolder** – path to the folder or tarball containing the smodels (python) output files
- **slhaFolder** – path to the folder or tarball containing the SLHA input files
- **parameterFile** – path to the file containing the plotting definitions
- **modelFile** – path to the model file, e.g smodels/share/models/mssm.py

display()
display the pages, works in jupyter notebooks only

editSlhaInformation()
Edits slha_hover_information,ctau_hover_information,BR_hover_information,variable_x,variable_y if they are defined as a list. The function transforms it in a dict whose keys are the object names

fillWith(*smodelsOutput*, *slhaData*)
Fill the dictionary (data_dict) with the desired data from the smodels output dictionary (smodelsDict) and the pyslha.Doc object slhaData

getParticleName(*pdg*)
looks for the particle label in the model.py file

initializeDataDict()
Initializes an empty dictionary with the plotting options.

loadData(*npoints=-1*)
Reads the data from the smodels and SLHA folders. If npoints > 0, it will limit the number of points in the plot to npoints.

Parameters **npoints** – Number of points to be plotted (int). If < 0, all points will be used.

loadModelFile()
Reads the parameters from the plotting parameter file.

loadParameters()
Reads the parameters from the plotting parameter file.

plot(*outFolder*, *indexfile='plots.html'*)
Uses the data in self.data_dict to produce the plots.

Parameters

- **outFolder** – Path to the output folder.
- **indexfile** – name of entry webpage

rmFiles(*flist*)
remove files in flist

tools.interactivePlots.**main**(*args*, *indexfile='index.html'*)
Create the interactive plots using the input from argparse

Parameters **args** – argparser.Namespace object containing the options for the plotter

Main interface for the interactive-plots.

Parameters

- **smodelsFolder** – Path to the folder or tarball containing the SModelS python output
- **slhaFolder** – Path to the folder or tarball containing the SLHA files corresponding to the SModelS output
- **parameters** – Path to the parameter file setting the options for the interactive plots
- **npoints** – Number of points used to produce the plot. If -1, all points will be used.
- **verbosity** – Verbosity of the output (debug,info,warning,error)
- **indexfile** – name of the starting web page (index.html)

Returns True if the plot creation was successfull

tools.interactivePlotsHelpers module

class tools.interactivePlotsHelpers.**Filler**(*plotter*, *smodelsOutput*, *slhaData*)
Bases: object

A class with the functions required to fill the data dictionary to produce the plots

getBR()
Gets the requested branching ratios from the slha file, that will go into de hover.

getCtau()
Computes the requested ctaus, that will go into de hover.

getExpres()
Extracts the Expres info from the .py output. If requested, the data will be appended on each corresponding list

getMaxMissingTopology()
Extracts the missing topology with the largest cross section

getMaxMissingTopologyXsection()
Extracts the cross section of the missing topology with the largest cross section

getOutsideGrid()
Extracts the outside grid info from the .py output. If requested, the data will be appended on each corresponding list.

getParticleName(*pdg*)
looks for the particle label in the model.py file

getSlhaData(*variable_x*, *variable_y*)
fills data dict with slha data

getSlhaHoverInfo()
Gets the requested slha info from each slha file, to fill the hover.

getSmodelSData()
fills data dict with smodels data

getTotalMissingDisplaced()
Extracts the Total cross section from missing displaced topologies

getTotalMissingPrompt()
Extracts the Total cross section from missing prompt topologies

getTotalMissingXsec()
Extracts the total crossection from missing topologies.

getVariable(*variable*)
Gets the variable from the slha file.

openSMParticles()
Loads SMparticles.py to parse over SM pdg-labels

truncate(*number*)
truncate float to 3 decimal places

class tools.interactivePlotsHelpers.**PlotlyBackend**(*master*, *path_to_plots*)
Bases: object

DataFrameExcludedNonexcluded()
Generate sub data frames for excluded and non-excluded points

GetXyAxis()
Retrieves the names of the x and y axis variables.

SeparateContDiscPlots()
Generate sub lists of plots with discrete and conitnuous z axis variables.

createIndexHtml()
Fills the index.html file with links to the interactive plots.

fillHover()
Generates the text of the hover, according to users's requests.

makeContinuousPlots(*data_frame*, *data_selection*)
Generate plots with continuous z axis variables, using all data points

makeDataFrame()
Transform the main dictionary in a data frame.

makeDiscretePlots(*data_frame*, *data_selection*)
Generate plots with discrete z axis variables, using all data points

makePlots(*indexfile*)
Uses the data in self.data_dict to produce the plots.

Parameters **outFolder** – Path to the output folder.

plotDescription()
Generate a description for each plot.

refiningVariableNames()
Redifining the output variable names to html format

tools.interactivePlotsHelpers.**getEntry**(*inputDict*, **keys*)
Get entry key in dictionary inputDict. If a list of keys is provided, it will assumed nested dictionaries (e.g. key1,key2 will return inputDict[key1][key2]).

tools.interactivePlotsHelpers.**getSlhaData**(*slhaFile*)
Uses pyslha to read the SLHA file. Return a pyslha.Doc objec, if successful.

tools.interactivePlotsHelpers.**getSlhaFile**(*smodelsOutput*)
Returns the file name of the SLHA file corresponding to the output in smodelsDict

tools.interactivePlotsHelpers.**importPythonOutput**(*smodelsFile*)
Imports the smodels output from each .py file.

tools.interactivePlotsHelpers.**outputStatus**(*smodelsDict*)
Check the smodels output status in the file, if it's -1, it will append 'none' to each list in the dictionary.

tools.ioObjects module

class tools.ioObjects.**FileStatus**
Bases: object

Object to run several checks on the input file. It holds an LheStatus (SlhaStatus) object if inputType = lhe (slha)

checkFile(*inputFile*)
Run checks on the input file.

Parameters **inputFile** – path to input file

class tools.ioObjects.**LheStatus**(*filename*)
Bases: object

Object to check if input lhe file contains errors.

Variables ***filename*** – path to input LHE file

evaluateStatus()
run status check

class tools.ioObjects.**OutputStatus**(*status*, *inputFile*, *parameters*, *databaseVersion*)

Bases: object

Object that holds all status information and has a predefined printout.

Initialize output. If one of the checks failed, exit.

Parameters

- **status** – status of input file
- **inputFile** – input file name
- **parameters** – input parameters
- **databaseVersion** – database version (string)

addWarning(*warning*)

Append warning to warnings.

Parameters **warning** – warning to be appended

updateSLHAStatus(*status*)

Update SLHA status.

Parameters **status** – new SLHA status flag

updateStatus(*status*)

Update status.

Parameters **status** – new status flag

class tools.ioObjects.**SlhaStatus**(*filename*, *findMissingDecayBlocks=True*, *findIllegalDecays=False*, *checkXsec=True*)

Bases: object

An instance of this class represents the status of an SLHA file. The output status is: = 0 : the file is not checked, = 1: the check is ok = -1: case of a physical problem, e.g. charged LSP, = -2: case of formal problems, e.g. no cross sections

Parameters

- **filename** – path to input SLHA file
- **findMissingDecayBlocks** – if True add a warning for missing decay blocks
- **findIllegalDecays** – if True check if all decays are kinematically allowed
- **checkXsec** – if True check if SLHA file contains cross sections
- **findLonglived** – if True find stable charged particles and displaced vertices

emptyDecay(*pid*)

Check if any decay is missing for the particle with pid

Parameters **pid** – PID number of particle to be checked

Returns True if the decay block is missing or if it is empty, None otherwise

evaluateStatus()

Get status summary from all performed checks.

Returns a status flag and a message for explanation

findIllegalDecay(*findIllegal*)

Find decays for which the sum of daughter masses excels the mother mass

Parameters **findIllegal** – True if check should be run

Returns status flag and message

findMissingDecayBlocks(*findMissingBlocks*)
For all non-SMpdgs particles listed in mass block, check if decay block is written

Returns status flag and message

hasXsec(*checkXsec*)
Check if XSECTION table is present in the slha file.

Parameters **checkXsec** – set True to run the check

Returns status flag, message

read()
Get pyslha output object.

tools.lheChecks module

tools.lheChecks.**main**(*args*)

tools.modelTester module

tools.modelTester.**checkForSemicolon**(*strng*, *section*, *var*)

tools.modelTester.**getAllInputFiles**(*inFile*)
Given inFile, return list of all input files

Parameters **inFile** – Path to input file or directory containing input files

Returns List of all input files, and the directory name

tools.modelTester.**getParameters**(*parameterFile*)
Read parameter file, exit in case of errors

Parameters **parameterFile** – Path to parameter File

Returns ConfigParser read from parameterFile

tools.modelTester.**loadDatabase**(*parser*, *db*)
Load database

Parameters

- **parser** – ConfigParser with path to database
- **db** – binary database object. If None, then database is loaded, according to databasePath. If True, then database is loaded, and text mode is forced.

Returns database object, database version

tools.modelTester.**loadDatabaseResults**(*parser*, *database*)
Load database entries specified in parser

Parameters

- **parser** – ConfigParser, containing analysis and txnames selection
- **database** – Database object

Returns List of experimental results

tools.modelTester.**runSetOfFiles**(*inputFiles*, *outputDir*, *parser*, *databaseVersion*, *listOfExpRes*, *timeout*, *development*, *parameterFile*)

Loop over all input files in inputFiles with testPoint

Parameters

- **inputFiles** – list of input files to be tested
- **outputDir** – path to directory where output is be stored
- **parser** – ConfigParser storing information from parameter.ini file
- **databaseVersion** – Database version (printed to output file)
- **listOfExpRes** – list of ExpResult objects to be considered
- **development** – turn on development mode (e.g. no crash report)
- **parameterFile** – parameter file, for crash reports

Returns printers output

tools.modelTester.**runSingleFile**(*inputFile*, *outputDir*, *parser*, *databaseVersion*, *listOfExpRes*, *timeout*, *development*, *parameterFile*)

Call testPoint on inputFile, write crash report in case of problems

Parameters

- **inputFile** – path to input file
- **outputDir** – path to directory where output is be stored
- **parser** – ConfigParser storing information from parameter.ini file
- **databaseVersion** – Database version (printed to output file)
- **listOfExpRes** – list of ExpResult objects to be considered
- **crashReport** – if True, write crash report in case of problems
- **timeout** – set a timeout for one model point (0 means no timeout)

Returns output of printers

tools.modelTester.**setExperimentalFlag**(*parser*)

set the experimental flag, if options:experimental = True

tools.modelTester.**testPoint**(*inputFile*, *outputDir*, *parser*, *databaseVersion*, *listOfExpRes*)

Test model point defined in input file (running decomposition, check results, test coverage)

Parameters

- **inputFile** – path to input file
- **outputDir** – path to directory where output is be stored
- **parser** – ConfigParser storing information from parameters file
- **databaseVersion** – Database version (printed to output file)
- **listOfExpRes** – list of ExpResult objects to be considered

Returns output of printers

tools.modelTester.**testPoints**(*fileList*, *inDir*, *outputDir*, *parser*, *databaseVersion*, *listOfExpRes*, *timeout*, *development*, *parameterFile*)

Loop over all input files in fileList with testPoint, using ncpus CPUs defined in parser

Parameters

- **fileList** – list of input files to be tested
- **inDir** – path to directory where input files are stored
- **outputDir** – path to directory where output is stored
- **parser** – ConfigParser storing information from parameter.ini file
- **databaseVersion** – Database version (printed to output files)
- **listOfExpRes** – list of ExpResult objects to be considered
- **timeout** – set a timeout for one model point (0 means no timeout)
- **development** – turn on development mode (e.g. no crash report)
- **parameterFile** – parameter file, for crash reports

Returns printer(s) output, if not run in parallel mode

tools.nllFastWrapper module

class tools.nllFastWrapper.**NllFastWrapper**(*sqrts*, *nllfastVersion*, *testParams*, *testCondition*)
Bases: smodels.tools.wrapperBase.WrapperBase

An instance of this class represents the installation of nllfast.

Parameters

- **sqrts** – sqrt of s, in TeV, as an integer,
- **nllfastVersion** – version of the nllfast tool
- **testParams** – what are the test params we need to run things with?
- **testCondition** – the line that should be the last output line when running executable

SrcPath the path of the source code, for compilation

getKfactorsFor(*pIDs*, *slhafile*, *pdf='cteq'*)
Read the NLLfast grid and returns a pair of k-factors (NLO and NLL) for the PIDs pair.

Returns k-factors = None, if NLLfast does not contain the process; uses the slhafile to obtain the SUSY spectrum.

class tools.nllFastWrapper.**NllFastWrapper13**
Bases: *tools.nllFastWrapper.NllFastWrapper*

An instance of this class represents the installation of nllfast 8.

class tools.nllFastWrapper.**NllFastWrapper7**
Bases: *tools.nllFastWrapper.NllFastWrapper*

An instance of this class represents the installation of nllfast 7.

class tools.nllFastWrapper.**NllFastWrapper8**
Bases: *tools.nllFastWrapper.NllFastWrapper*

An instance of this class represents the installation of nllfast 8.

tools.physicsUnits module

tools.printer module

class tools.printer.**BasicPrinter**(*output*, *filename*)
Bases: object

Super class to handle the basic printing methods

addObj(*obj*)
Adds object to the Printer.

Parameters **obj** – A object to be printed. Must match one of the types defined in formatObj

Returns True if the object has been added to the output. If the object does not belong to the pre-defined printing list toPrint, returns False.

filename

flush()
Format the objects added to the output, print them to the screen or file and remove them from the printer.

mkdir()
create directory to file, if necessary

openOutFile(*filename*, *mode*)
creates and opens a data sink, creates path if needed

setOptions(*options*)
Store the printer specific options to control the output of each printer. Each option is stored as a printer attribute.

Parameters **options** – a list of (option,value) for the printer.

class tools.printer.**MPrinter**
Bases: object

Master Printer class to handle the Printers (one printer/output type)

addObj(*obj*)
Adds the object to all its Printers:

Parameters **obj** – An object which can be handled by the Printers.

flush()
Ask all printers to write the output and clear their cache. If the printers return anything other than None, we pass it on.

setOutPutFiles(*filename*, *silent=False*)
Set the basename for the output files. Each printer will use this file name appended of the respective extension (i.e. .py for a python printer, .smodels for a summary printer,...)

Parameters

- **filename** – Input file name
- **silent** – dont comment removing old files

setPrinterOptions(*parser*)
Define the printer types and their options.

Parameters **parser** – ConfigParser storing information from the parameters file

class tools.printer.**PyPrinter**(*output='stdout'*, *filename=None*)
Bases: *tools.printer.BasicPrinter*

Printer class to handle the printing of one single pythonic output

flush()
Write the python dictionaries generated by the object formatting to the defined output

setOutPutFile(*filename*, *overwrite=True*, *silent=False*)
Set the basename for the text printer. The output filename will be filename.py. :param filename: Base filename :param overwrite: If True and the file already exists, it will be removed. :param silent: dont comment removing old files

class tools.printer.**SLHAPrinter**(*output='file'*, *filename=None*)
Bases: *tools.printer.TxTPrinter*

Printer class to handle the printing of slha format summary output. It uses the facilities of the TxTPrinter.

setOutPutFile(*filename*, *overwrite=True*, *silent=False*)
Set the basename for the text printer. The output filename will be filename.smodels. :param filename: Base filename :param overwrite: If True and the file already exists, it will be removed. :param silent: dont comment removing old files

class tools.printer.**SummaryPrinter**(*output='stdout'*, *filename=None*)
Bases: *tools.printer.TxTPrinter*

Printer class to handle the printing of one single summary output. It uses the facilities of the TxTPrinter.

setOutPutFile(*filename*, *overwrite=True*, *silent=False*)
Set the basename for the text printer. The output filename will be filename.smodels. :param filename: Base filename :param overwrite: If True and the file already exists, it will be removed. :param silent: dont comment removing old files

class tools.printer.**TxTPrinter**(*output='stdout'*, *filename=None*)
Bases: *tools.printer.BasicPrinter*

Printer class to handle the printing of one single text output

setOutPutFile(*filename*, *overwrite=True*, *silent=False*)
Set the basename for the text printer. The output filename will be filename.log.

Parameters

- **filename** – Base filename
- **overwrite** – If True and the file already exists, it will be removed.
- **silent** – dont comment removing old files

class tools.printer.**XmlPrinter**(*output='stdout'*, *filename=None*)
Bases: *tools.printer.PyPrinter*

Printer class to handle the printing of one single XML output

convertToElement(*pyObj*, *parent*, *tag=''*)
Convert a python object (list,dict,string,...) to a nested XML element tree. :param pyObj: python object (list,dict,string...) :param parent: XML Element parent :param tag: tag for the daughter element

flush()
Get the python dictionaries generated by the object formatting to the defined output and convert to XML

setOutPutFile(*filename*, *overwrite=True*, *silent=False*)
Set the basename for the text printer. The output filename will be filename.xml. :param filename: Base

filename :param overwrite: If True and the file already exists, it will be removed. :param silent: dont comment removing old files

tools.printer.**getInfoFromPython**(*output*)
Retrieves information from the python output

Parameters **output** – output (dictionary)

Returns list of r-values,r-expected and analysis IDs. None if no results are found.

tools.printer.**getInfoFromSLHA**(*output*)
Retrieves information from the SLHA output

Parameters **output** – output (string)

Returns list of r-values,r-expected and analysis IDs. None if no results are found.

tools.printer.**getInfoFromSummary**(*output*)
Retrieves information from the summary output

Parameters **output** – output (string)

Returns list of r-values,r-expected and analysis IDs. None if no results are found.

tools.printer.**getSummaryFrom**(*output*, *ptype*)
Retrieves information about the output according to the printer type (slha,python or summary)

Parameters

- **output** – output (dictionary for ptype=python or string for ptype=slha/summary)
- **ptype** – Printer type (slha, python or summary)

Returns Dictionary with the output information

tools.printer.**printScanSummary**(*outputDict*, *outputFile*)
Method for creating a simple summary of the results when running SModelS over multiple files.

Parameters

- **outputDict** – A dictionary with filenames as keys and the master printer flush dictionary as values.
- **outputFile** – Path to the summary file to be written.

tools.proxyDBCreator module

class tools.proxyDBCreator.**ProxyDBCreater**(*inputfile*, *rundir*, *verbose='info'*)
Bases: object

create(*servername*, *serverport*)

pprint(**args*)

run(*really*)
now run the server :param really: if False, then only write out command

store(*outputfile*)
store the outputfile

symlink()
set a symlink from self.outputfile to default.pcl

tools.proxyDBCreator.**main**(*args*)

tools.pyhfInterface module

class tools.pyhfInterface.**PyhfData**(*nsignals*, *inputJsons*)
Bases: object

Holds data for use in pyhf :ivar nsignals: signal predictions list divided into sublists, one for each json file :ivar inputJsons: list of json instances :ivar nWS: number of workspaces = number of json files

checkConsistency()
Check various inconsistencies of the PyhfData attributes

Parameters **zeroSignalsFlag** – boolean identifying if all SRs of a single json are empty

getWSInfo()
Getting informations from the json files

Variables **channelsInfo** – list of dictionaries (one dictionary for each json file) containing useful information about the json files - :key signalRegions: list of dictonaries with 'json path' and 'size' (number of bins) of the 'signal regions' channels in the json files - :key otherRegions: list of strings indicating the path to the control and validation region channels

class tools.pyhfInterface.**PyhfUpperLimitComputer**(*data*, *cl=0.95*)
Bases: object

Class that computes the upper limit using the jsons files and signal informations in the *data* instance of *PyhfData*

Parameters

- **data** – instance of *PyhfData* holding the signals information
- **cl** – confdence level at which the upper limit is desired to be computed

Variables

- **data** – created from :param data:
- **nsignals** – signal predictions list divided into sublists, one for each json file
- **inputJsons** – list of input json files as python json instances
- **channelsInfo** – list of channels information for the json files
- **zeroSignalsFlag** – list boolean flags in case all signals are zero for a specific json
- **nWS** – number of workspaces = number of json files
- **patches** – list of patches to be applied to the inputJsons as python dictionary instances
- **workspaces** – list of workspaces resulting from the patched inputJsons
- **cl** – created from :param cl:
- **scale** – scale that is applied to the signal predictions, dynamically changes throughout the upper limit calculation
- **alreadyBeenThere** – boolean flag that identifies when the :ivar nsignals: accidentally passes twice at two identical values

checkPyhfVersion()
check the pyhf version, currently we need 0.6.1+

chi2(*workspace_index=None*)
Returns the chi square

likelihood(*workspace_index=None*)
Returns the value of the likelihood. Inspired by the *pyhf.infer.mle* module but for non-log likelihood

patchMaker()
Method that creates the list of patches to be applied to the *self.inputJsons* workspaces, one for each region given the *self.nsignals* and the informations available in *self.channelsInfo* and the content of the *self.inputJsons* NB: It seems we need to include the change of the "modifiers" in the patches as well

Returns the list of patches, one for each workspace

rescale(*factor*)
Rescales the signal predictions (self.nsignals) and processes again the patches and workspaces

Returns updated list of patches and workspaces (self.patches and self.workspaces)

ulSigma(*expected=False*, *workspace_index=None*)

Compute the upper limit on the signal strength modifier with:

- by default, the combination of the workspaces contained into self.workspaces
- if workspace_index is specified, self.workspace[workspace_index] (useful for computation of the best upper limit)

Parameters

- **expected** –
  - if set to *True*: uses expected SM backgrounds as signals
  - else: uses *self.nsignals*
- **workspace_index** –
  - if different from *None*: index of the workspace to use for upper limit
  - else: all workspaces are combined

Returns the upper limit at *self.cl* level (0.95 by default)

welcome()
greet the world

wsMaker()
Apply each region patch (self.patches) to his associated json (self.inputJsons) to obtain the complete workspaces

Returns the list of patched workspaces

tools.pyhfInterface.**getLogger**()
Configure the logging facility. Maybe adapted to fit into your framework.

tools.pythia6Wrapper module

class tools.pythia6Wrapper.**Pythia6Wrapper**(*configFile='<install>/smodels/etc/pythia.card'*, *executablePath='<install>/smodels/lib/pythia6/pythia_lhe'*, *srcPath='<install>/smodels/lib/pythia6/'*)
Bases: smodels.tools.wrapperBase.WrapperBase

An instance of this class represents the installation of pythia6. nevents keeps track of how many events we run. For each event we only allow a certain computation time: if self.secondsPerEvent * self.nevents > CPU time, we terminate Pythia.

Parameters

- **configFile** – Location of the config file, full path; copy this file and provide tools to change its content and to provide a template
- **executablePath** – Location of executable, full path (pythia_lhe)
- **srcPath** – Location of source code

checkFileExists(*inputFile*)
Check if file exists, raise an IOError if it does not.

Returns absolute file name if file exists.

replaceInCfgFile(*replacements={'NEVENTS': 10000, 'SQRTS': 8000}*)
Replace strings in the config file by other strings, similar to setParameter.

This is introduced as a simple mechanism to make changes to the parameter file.

Parameters **replacements** – dictionary of strings and values; the strings will be replaced with the values; the dictionary keys must be strings present in the config file

run(*slhafile*, *lhefile=None*, *unlink=True*)
Execute pythia_lhe with n events, at sqrt(s)=sqrts.

Parameters

- **slhafile** – input SLHA file
- **lhefile** – option to write LHE output to file; if None, do not write output to disk. If lhe file exists, use its events for xsecs calculation.
- **unlink** – Clean up temp directory after running pythia

Returns List of cross sections

setParameter(*param='MSTP(163)'*, *value=6*)
Modifies the config file, similar to .replaceInCfgFile.

It will set param to value, overwriting possible old values.

unlink(*unlinkdir=True*)
Remove temporary files.

Parameters **unlinkdir** – remove temp directory completely

tools.pythia8Wrapper module

class tools.pythia8Wrapper.**Pythia8Wrapper**(*configFile='<install>/smodels/etc/pythia8.cfg'*, *executablePath='<install>/smodels/lib/pythia8/pythia8.exe'*, *srcPath='<install>/smodels/lib/pythia8/'*)
Bases: smodels.tools.wrapperBase.WrapperBase

An instance of this class represents the installation of pythia8.

Parameters

- **configFile** – Location of the config file, full path; copy this file and provide tools to change its content and to provide a template
- **executablePath** – Location of executable, full path (pythia8.exe)
- **srcPath** – Location of source code

checkFileExists(*inputFile*)
Check if file exists, raise an IOError if it does not.

Returns absolute file name if file exists.

chmod()
chmod 755 on pythia executable, if it exists. Do nothing, if it doesnt exist.

run(*slhaFile*, *lhefile=None*, *unlink=True*)
Run pythia8.

Parameters

- **slhaFile** – SLHA file
- **lhefile** – option to write LHE output to file; if None, do not write output to disk. If lhe file exists, use its events for xsecs calculation.
- **unlink** – clean up temporary files after run?

Returns List of cross sections

unlink(*unlinkdir=True*)
Remove temporary files.

Parameters **unlinkdir** – remove temp directory completely

tools.pythia8particles module

tools.reweighting module

tools.reweighting.**calculateProbabilities**(*width*, *Leff_inner*, *Leff_outer*)
The fraction of prompt and displaced decays are defined as:

F_long = exp(-totalwidth*l_outer/gb_outer) F_prompt = 1 - exp(-totaltotalwidth*l_inner/gb_inner) F_displaced = 1 - F_prompt - F_long

Parameters

- **Leff_inner** – is the effective inner radius of the detector, given in meters
- **Leff_outer** – is the effective outer radius of the detector, given in meters
- **width** – particle width for which probabilities should be calculated (in GeV)

Returns Dictionary with the probabilities for the particle not to decay (in the detector), to decay promptly or displaced.

tools.reweighting.**defaultEffReweight**(*element*, *Leff_inner=None*, *Leff_outer=None*, *minWeight=1e-10*)
Computes the lifetime reweighting factor for the element efficiency based on the lifetimes of all intermediate particles and the last stable odd-particle appearing in the element. The fraction corresponds to the fraction of decays corresponding to prompt decays to all intermediate BSM particles and to a long-lived decay (outside the detector) to the final BSM state.

Parameters

- **element** – Element object or nested list of widths
- **minWeight** – Lower cut for the reweighting factor. Any value below this will be taken to be zero.
- **Leff_inner** – is the effective inner radius of the detector, given in meters. If None, use default value.

- **Leff_outer** – is the effective outer radius of the detector, given in meters. If None, use default value.

Returns Reweight factor (float)

tools.reweighting.**defaultULReweight**(*element*, *Leff_inner=None*, *Leff_outer=None*)
Computes the lifetime reweighting factor for the element upper limit based on the lifetimes of all intermediate particles and the last stable odd-particle appearing in the element. The fraction corresponds to the fraction of decays corresponding to prompt decays to all intermediate BSM particles and to a long-lived decay (outside the detector) to the final BSM state.

Parameters

- **element** – Element object
- **Leff_inner** – is the effective inner radius of the detector, given in meters. If None, use default value.
- **Leff_outer** – is the effective outer radius of the detector, given in meters. If None, use default value.

Returns Reweight factor (float)

tools.reweighting.**reweightFactorFor**(*element*, *resType='prompt'*, *Leff_inner=None*, *Leff_outer=None*)
Computer the reweighting factor for the element according to the experimental result type. Currently only two result types are supported: 'prompt' and 'displaced'. If resultType = 'prompt', returns the reweighting factor for all decays in the element to be prompt and the last odd particle to be stable. If resultType = 'displaced', returns the reweighting factor for ANY decay in the element to be displaced and no long-lived decays and the last odd particle to be stable. Not that the fraction of "long-lived (meta-stable) decays" is usually included in topologies where the meta-stable particle appears in the final state. Hence it should not be included in the prompt or displaced fractions.

Parameters

- **element** – Element object
- **resType** – Type of result to compute the reweight factor for (either 'prompt' or 'displaced')
- **Leff_inner** – is the effective inner radius of the detector, given in meters. If None, use default value.
- **Leff_outer** – is the effective outer radius of the detector, given in meters. If None, use default value.

Returns probabilities (depending on types of decay within branch), branches (with different labels depending on type of decay)

tools.runSModelS module

tools.runSModelS.**main**()

tools.runSModelS.**run**(*inFile*, *parameterFile*, *outputDir*, *db*, *timeout*, *development*)
Provides a command line interface to basic SModelS functionalities.

Parameters

- **inFile** – input file name (either a SLHA or LHE file) or directory name (path to directory containing input files)

- **parameterFile** – File containing the input parameters (default = smodels/etc/parameters_default.ini)
- **outputDir** – Output directory to write a summary of results to
- **db** – supply a smodels.experiment.databaseObj.Database object, so the database doesn't have to be loaded anymore. Will render a few parameters in the parameter file irrelevant. If None, load the database as described in parameterFile, If True, force loading the text database.
- **timeout** – set a timeout for one model point (0 means no timeout)
- **development** – turn on development mode (e.g. no crash report)

tools.runtime module

tools.runtime.**experimentalFeatures**()
a simple boolean flag to turn experimental features on/off, can be turned on and off via options:experimental in parameters.ini.

tools.runtime.**filetype**(*filename*)
obtain information about the filetype of an input file, currently only used to discriminate between slha and lhe files.

Returns filetype as string("slha" or "lhe"), None if file does not exist, or filetype is unknown.

tools.runtime.**nCPUs**()
obtain the number of CPU cores on the machine, for several platforms and python versions.

tools.simplifiedLikelihoods module

class tools.simplifiedLikelihoods.**Data**(*observed*, *backgrounds*, *covariance*, *third_moment=None*, *nsignal=None*, *name='model'*, *deltas_rel=0.2*)
Bases: object

A very simple observed container to collect all the data needed to fully define a specific statistical model

Parameters

- **observed** – number of observed events per dataset
- **backgrounds** – expected bg per dataset
- **covariance** – uncertainty in background, as a covariance matrix
- **nsignal** – number of signal events in each dataset
- **name** – give the model a name, just for convenience
- **deltas_rel** – the assumed relative error on the signal hypotheses. The default is 20%.

convert(*obj*)
Convert object to numpy arrays. If object is a float or int, it is converted to a one element array.

correlations()
Correlation matrix, computed from covariance matrix. Convenience function.

diagCov()
Diagonal elements of covariance matrix. Convenience function.

isLinear()
Statistical model is linear, i.e. no quadratic term in poissonians

isScalar(*obj*)
Determine if obj is a scalar (float or int)

sandwich()
Sandwich product

signals(*mu*)
Returns the number of expected signal events, for all datasets, given total signal strength mu.

Parameters **mu** – Total number of signal events summed over all datasets.

totalCovariance(*nsig*)
get the total covariance matrix, taking into account also signal uncertainty for the signal hypothesis <nsig>. If nsig is None, the predefined signal hypothesis is taken.

var_s(*nsig=None*)
The signal variances. Convenience function.

Parameters **nsig** – If None, it will use the model expected number of signal events, otherwise will return the variances for the input value using the relative signal uncertainty defined for the model.

zeroSignal()
Is the total number of signal events zero?

class tools.simplifiedLikelihoods.**LikelihoodComputer**(*data*, *ntoys=30000*)
Bases: object

Parameters

- **data** – a Data object.
- **ntoys** – number of toys when marginalizing

chi2(*nsig*, *marginalize=False*)
Computes the chi2 for a given number of observed events nobs given the predicted background nb, error on this background deltab, expected number of signal events nsig and the relative error on signal (deltas_rel). :param marginalize: if true, marginalize, if false, profile :param nsig: number of signal events :return: chi2 (float)

dLdMu(*mu*, *signal_rel*, *theta_hat*)
d (ln L)/d mu, if L is the likelihood. The function whose root gives us muhat, i.e. the mu that maximizes the likelihood.

Parameters

- **mu** – total number of signal events
- **signal_rel** – array with the relative signal strengths for each dataset (signal region)
- **theta_hat** – array with nuisance parameters

debug_mode = False

findMuHat(*signal_rel*)
Find the most likely signal strength mu given the relative signal strengths in each dataset (signal region).

Parameters **signal_rel** – array with relative signal strengths

Returns mu_hat, the maximum likelihood estimate of mu

findThetaHat(*nsig*)
Compute nuisance parameter theta that maximizes our likelihood (poisson*gauss).

getSigmaMu(*signal_rel*)
Get a rough estimate for the variance of mu around mu_max.

Parameters **signal_rel** – array with relative signal strengths in each dataset (signal region)

getThetaHat(*nobs*, *nb*, *nsig*, *covb*, *max_iterations*)
Compute nuisance parameter theta that maximizes our likelihood (poisson*gauss).

likelihood(*nsig*, *marginalize=False*, *nll=False*)
compute likelihood for nsig, profiling the nuisances :param marginalize: if true, marginalize, if false, profile :param nll: return nll instead of likelihood

marginalizedLLHD1D(*nsig*, *nll*)
Return the likelihood (of 1 signal region) to observe nobs events given the predicted background nb, error on this background (deltab), expected number of signal events nsig and the relative error on the signal (deltas_rel).

Parameters

- **nsig** – predicted signal (float)
- **nobs** – number of observed events (float)
- **nb** – predicted background (float)
- **deltab** – uncertainty on background (float)

Returns likelihood to observe nobs events (float)

marginalizedLikelihood(*nsig*, *nll*)
compute the marginalized likelihood of observing nsig signal event

nll(*theta*)
probability, for nuicance parameters theta, as a negative log likelihood.

nllHess(*theta*)
the Hessian of nll as a function of the thetas. Makes it easier to find the maximum likelihood.

nllprime(*theta*)
the derivative of nll as a function of the thetas. Makes it easier to find the maximum likelihood.

probMV(*nll*, *theta*)
probability, for nuicance parameters theta :params nll: if True, compute negative log likelihood

profileLikelihood(*nsig*, *nll*)
compute the profiled likelihood for nsig. Warning: not normalized. Returns profile likelihood and error code (0=no error)

class tools.simplifiedLikelihoods.**UpperLimitComputer**(*ntoys=30000*, *cl=0.95*)
Bases: object

Parameters

- **ntoys** – number of toys when marginalizing
- **cl** – desired quantile for limits

debug_mode = False

ulSigma(*model*, *marginalize=False*, *toys=None*, *expected=False*, *trylasttime=False*)

upper limit obtained from the defined Data (using the signal prediction for each signal regio/dataset), by using the q_mu test statistic from the CCGV paper (arXiv:1007.1727).

Params marginalize if true, marginalize nuisances, else profile them

Params toys specify number of toys. Use default is none

Params expected compute the expected value, not the observed.

Params trylasttime if True, then dont try extra

Returns upper limit on *production* xsec (efficiencies unfolded)

tools.simplifiedLikelihoods.**getLogger**()
Configure the logging facility. Maybe adapted to fit into your framework.

tools.slhaChecks module

tools.slhaChecks.**main**(*args*)

tools.smodelsLogging module

class tools.smodelsLogging.**ColorizedStreamHandler**(*stream=None*)
Bases: logging.StreamHandler

format(*record*)
Format the specified record.

If a formatter is set, use it. Otherwise, use the default formatter for the module.

should_color()

tools.smodelsLogging.**getLogLevel**(*asString=False*)
obtain the current log level. :params asString: return string, not number.

tools.smodelsLogging.**getLogger**()

tools.smodelsLogging.**setLogLevel**(*level*)
set the log level of the central logger. can either be directly an integer (e.g. logging.DEBUG), or "debug", "info", "warning", or "error".

tools.smodelsTools module

tools.smodelsTools.**main**()

tools.statistics module

tools.statistics.**chi2FromLimits**(*likelihood*, *expectedUpperLimit*)
compute the chi2 value from a likelihood (convenience function).

tools.statistics.**deltaChi2FromLlhd**(*likelihood*)
compute the delta chi2 value from a likelihood (convenience function)

tools.statistics.**likelihoodFromLimits**(*upperLimit*, *expectedUpperLimit*, *nsig*, *nll=False*)
computes the likelihood from an expected and an observed upper limit. :param upperLimit: observed upper limit, as a yield (i.e. unitless) :param expectedUpperLimit: expected upper limit, also as a yield :param nSig: number of signal events :param nll: if True, return negative log likelihood

Returns likelihood (float)

tools.statistics.**rvsFromLimits**(*upperLimit*, *expectedUpperLimit*, *n=1*)
Generates a sample of random variates, given expected and observed likelihoods. The likelihood is modelled as a truncated Gaussian.

Parameters

- **upperLimit** – observed upper limit, as a yield (i.e. unitless)
- **expectedUpperLimit** – expected upper limit, also as a yield
- **n** – sample size

Returns sample of random variates

tools.stringTools module

tools.stringTools.**cleanWalk**(*topdir*)
perform os.walk, but ignore all hidden files and directories

tools.stringTools.**concatenateLines**(*oldcontent*)
of all lines in the list "oldcontent", concatenate the ones that end with or ,

tools.timeOut module

exception tools.timeOut.**NoTime**(*value=None*)
Bases: Exception

The time out exception. Raised when the running time exceeds timeout

class tools.timeOut.**Timeout**(*sec*)
Bases: object

Timeout class using ALARM signal.

raise_timeout(**args*)

tools.toolBox module

class tools.toolBox.**ToolBox**
Bases: object

A singleton-like class that keeps track of all external tools. Intended to make installation and deployment easier.

Constructor creates the singleton.

add(*instance*)
Adds a tool by passing an instance to this method.

checkInstallation(*make=False*, *printit=True*, *longL=False*)
Checks if all tools listed are installed properly, returns True if everything is ok, False otherwise.

compile()
Tries to compile and install tools that are not yet marked as 'installed'.

get(*tool*, *verbose=True*)
Gets instance of tool from the toolbox.

initSingleton()
Initializes singleton instance (done only once for the entire class).

installationOk(*ok*)
Returns color coded string to signal installation issues.

listOfTools()
Returns a simple list with the tool names.

tools.toolBox.**main**(*args*)

tools.wrapperBase module

class tools.wrapperBase.**WrapperBase**
Bases: object

An instance of this class represents the installation of an external tool.

An external tool encapsulates a tool that is executed via commands.getoutput. The wrapper defines how the tool is tested for proper installation and how the tool is executed.

absPath(*path*)
Get the absolute path of <path>, replacing <install> with the installation directory.

basePath()
Get the base installation path.

checkInstallation(*compile=True*)
Checks if installation of tool is correct by looking for executable and executing it. If check is False and compile is True, then try and compile it.

Returns True, if everything is ok

chmod()
chmod 755 on executable, if it exists. Do nothing, if it doesnt exist.

compile()
Try to compile the tool.

complain()

defaulttempdir = '/tmp/'

installDirectory()

Returns the installation directory of the tool

pathOfExecutable()

Returns path of executable

tempDirectory()
Return the temporary directory name.

tools.wrapperBase.**ok**(*b*)

Returns 'ok' if b is True, else, return 'error'.

tools.xsecComputer module

class tools.xsecComputer.**ArgsStandardizer**
Bases: object

simple class to collect all argument manipulators

checkAllowedSqrtses(*order*, *sqrtses*)
check if the sqrtses are 'allowed'

checkNCPUs(*ncpus*, *inputFiles*)

getInputFiles(*args*)
geth the names of the slha files to run over

getOrder(*args*)
retrieve the order in perturbation theory from argument list

getPythiaVersion(*args*)

getSSMultipliers(*multipliers*)

getSqrtses(*args*)
extract the sqrtses from argument list

queryCrossSections(*filename*)

writeToFile(*args*)

class tools.xsecComputer.**XSecComputer**(*maxOrder*, *nevents*, *pythiaVersion*, *maycompile=True*)

Bases: object

cross section computer class, what else?

Parameters

- **maxOrder** – maximum order to compute the cross section, given as an integer if maxOrder == LO, compute only LO pythia xsecs if maxOrder == NLO, apply NLO K-factors from NLLfast (if available) if maxOrder == NLL, apply NLO+NLL K-factors from NLLfast (if available)
- **nevents** – number of events for pythia run
- **pythiaVersion** – pythia6 or pythia8 (integer)
- **maycompile** – if True, then tools can get compiled on-the-fly

addCommentToFile(*comment*, *slhaFile*)
add the optional comment to file

addHigherOrders(*sqrts*, *slhafile*)
add higher order xsecs

addMultipliersToFile(*ssmultipliers*, *slhaFile*)
add the signal strength multipliers to the SLHA file

addXSecToFile(*xsecs*, *slhafile*, *comment=None*, *complain=True*)
Write cross sections to an SLHA file.

Parameters

- **xsecs** – a XSectionList object containing the cross sections
- **slhafile** – target file for writing the cross sections in SLHA format
- **comment** – optional comment to be added to each cross section block
- **complain** – complain if there are already cross sections in file

applyMultipliers(*xsecs*, *ssmultipliers*)
apply the given multipliers to the cross sections

compute(*sqrts*, *slhafile*, *lhefile=None*, *unlink=True*, *loFromSlha=None*, *pythiacard=None*, *ssmultipliers=None*)

Run pythia and compute SUSY cross sections for the input SLHA file.

Parameters

- **sqrts** – sqrt{s} to run Pythia, given as a unum (e.g. 7.*TeV)
- **slhafile** – SLHA file
- **lhefile** – LHE file. If None, do not write pythia output to file. If file does not exist, write pythia output to this file name. If file exists, read LO xsecs from this file (does not run pythia).
- **unlink** – Clean up temp directory after running pythia
- **loFromSlha** – If True, uses the LO xsecs from the SLHA file to compute the higher order xsecs
- **pythiaCard** – Optional path to pythia.card. If None, uses smodels/etc/pythia.card
- **ssmultipliers** – optionally supply signal strengh multipliers, given as dictionary of the tuple of the mothers' pids as keys and multipliers as values, e.g { (1000001,1000021):1.1 }.

Returns XSectionList object

computeForBunch(*sqrtses*, *inputFiles*, *unlink*, *lOfromSLHA*, *tofile*, *pythiacard=None*, *ssmultipliers=None*)

compute xsecs for a bunch of slha files

computeForOneFile(*sqrtses*, *inputFile*, *unlink*, *lOfromSLHA*, *tofile*, *pythiacard=None*, *ssmultipliers=None*, *comment=None*)

Compute the cross sections for one file.

Parameters

- **sqrtses** – list of sqrt{s} tu run pythia, as a unum (e.g. [7*TeV])
- **inputFile** – input SLHA file to compute xsecs for
- **unlink** – if False, keep temporary files
- **lofromSLHA** – try to obtain LO xsecs from SLHA file itself
- **tofile** – False, True, "all": write results to file, if "all" also write lower xsecs to file.
- **pythiacard** – optionally supply your own runcard
- **ssmultipliers** – optionally supply signal strengh multipliers, given as dictionary of the tuple of the mothers' pids as keys and multipliers as values, e.g { (1000001,1000021):1.1 }.
- **comment** – an optional comment that gets added to the slha file.

Returns number of xsections that have been computed

getPythia()

returns the pythia tool that is configured to be used

match(*pids*, *theorypid*)

do the pids given by the user match the pids of the theorypred?

xsecToBlock(*xsec, inPDGs=(2212, 2212), comment=None, xsecUnit=1.00E+00 [pb]*)

Generate a string for a XSECTION block in the SLHA format from a XSection object.

Parameters

- **inPDGs** – defines the PDGs of the incoming states (default = 2212,2212)
- **comment** – is added at the end of the header as a comment
- **xsecUnit** – unit of cross sections to be written (default is pb). Must be a Unum unit.

tools.xsecComputer.**main**(*args*)

Module contents

share package

Subpackages

share.models package

Submodules

share.models.SMparticles module

share.models.dgmssm module

share.models.idm module

share.models.mssm module

share.models.nmssm module

Module contents

Module contents

Module contents

Indices and tables

- genindex
- modindex
- search

1.8 C++ Interface

Since v1.1.1, a simple C++ interface is provided, see the *smodels/cpp* directory in the source code.

Its usage is documented in run.cpp:

```
#include <SModelS.h>
#include <iostream>
#include <string>

/** example only */
```

(continues on next page)

(continued from previous page)

```
using namespace std;

int main( int argc, char * argv[] )
{
  /** initialise SModelS, load database. The first argument
   *  points to the SModelS config file,
   *  the second argument (``smodelsdir'') must point to the
   *  top-level directory of the SModelS installation */
  SModelS smodels ( "./parameters.ini", "../" );
  /** run over one single file or directory */
  int ret = smodels.run ( "test.slha" );
  /* return value, zero if successful */
  return ret;
}
```

A sample Makefile is also provided in the same directory. The python header files have to be installed; on debian-based systems the package name is *libpython3-dev*, on rpm-based systems it is usually called *python3-devel*.

2 Indices and tables

- genindex
- search

Python Module Index

W

X

Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z