SModelS Manual

Release 1.2.3

Federico Ambrogi, Juhi Dutta, Jan Heisig, Charanjit K. Khosa, Sabine Kraml, S

Apr 27, 2020

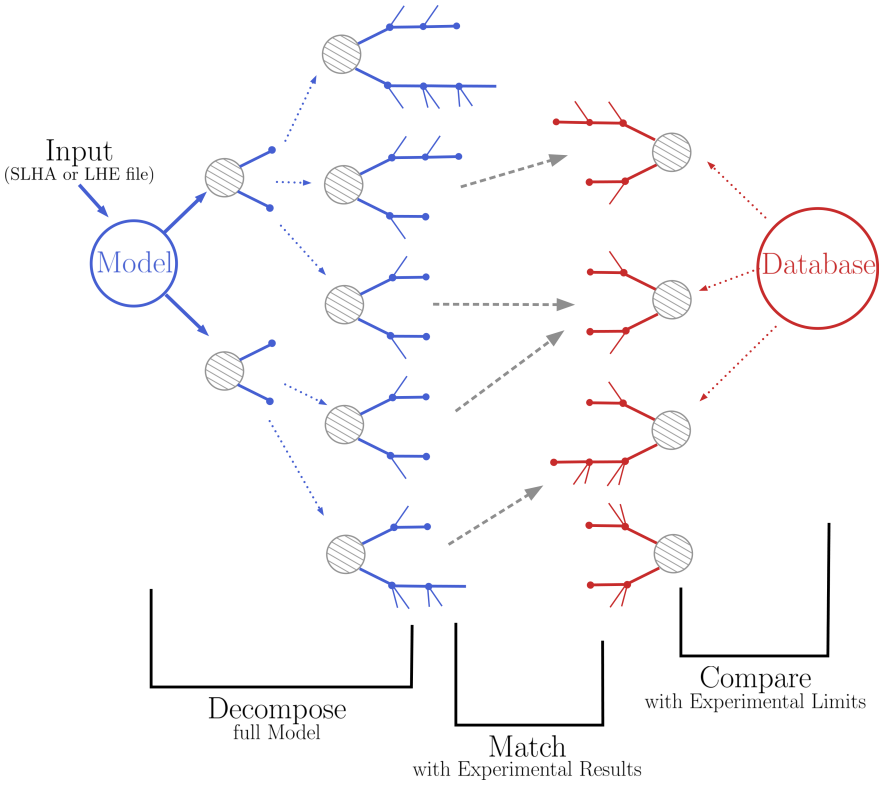
Contents

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 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. Our concrete implementation currently focusses on supersymmetry searches with missing energy, for which a large variety of SMS results from ATLAS and CMS are available. The main 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 check 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 1.2.3:

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

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:

- unum>=4.0.0
- numpy>=1.13.0
- argparse
- requests>=2.0.0
- docutils>=0.3
- scipy>=1.0.0
- pyslha>=3.1.0

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

SModelS v1.1.1 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 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

-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

[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

-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.
- *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 0.03 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/official113. The path "official" refers to the official database of your SModelS version – without fastlim; "official_fastlim" includes fastlim results. 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.
- *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.

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 import particlesLoader
from smodels.theory import slhaDecomposer,lheDecomposer
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
setLogLevel("info")
```

- *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.

```
# Set the path to the database
database = Database("official")
```

- *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*).

```
    #Define your model (list of rEven and rOdd particles)
    particlesLoader.load( 'smodels.share.models.mssm' ) #Make sure all the model␣
↪particles are up-to-date
```

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

```
    slhafile = 'inputFiles/slha/lightEWinos.slha'
    lhefile = 'inputFiles/lhe/gluino_squarks.lhe'
```

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

```
    sigmacut = 0.01 * fb
    mingap = 5. * GeV
```

- *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 (use slhaDecomposer for SLHA input or lheDecomposer for LHE␣
↪input)
    slhaInput = True
```

(continues on next page)

(continued from previous page)

```
    if slhaInput:
        toplist = slhaDecomposer.decompose(slhafile, sigmacut, doCompress=True,␣
↪doInvisible=True, minmassgap=mingap)
    else:
```

- *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 (if it exists):
    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="" )
        print( "\n\t\t\twith final states =",el.getFinalStates(),"\n\t\t\twith cross␣
↪section =",el.weight,"\n\t\t\tand masses = ",el.getMasses() )
```

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]']
                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:

```
    listOfExpRes = database.getExpResults()
```

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:

```
    nUL, nEM = 0, 0
    for exp in listOfExpRes:
        expType = exp.getValuesFor('dataType')[0]
        if expType == 'upperLimit':
            nUL += 1
        elif  expType == 'efficiencyMap':
            nEM += 1
    print( "\n Loaded Database with %i UL results and %i EM results " %(nUL,nEM) )
```

output:

```
Loaded Database with 55 UL results and 21 EM results
```

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

```
    for expResult in listOfExpRes:
        predictions = theoryPredictionsFor(expResult, toplist, combinedResults=False,␣
↪marginalize=False)
```

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

```
        for theoryPrediction in predictions:
            dataset = theoryPrediction.dataset
            datasetID = dataset.dataInfo.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
            print( "Condition Violation = ",theoryPrediction.conditions ) #Condition␣
↪violation values
```

output:

```
 ATLAS-SUSY-2015-06
-------------------------
Dataset =  SR5j
TxNames =  ['T1']
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]:5.19E-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:

```
            print( "UL for theory prediction = ",theoryPrediction.upperLimit )
```

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:

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

output:

```
r =  0.0029013935307768326
Chi2, likelihood= 2.3776849368423356 0.007169156710956845
```

- *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*:

```
    if rmax > 1.:
        print( "(The input model is likely excluded by %s)" %bestResult )
    else:
        print( "(The input model is not excluded by the simplified model results)" )
```

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:

```
    print( "Total cross section where we are outside the mass grid (fb): %10.3E\n"
↪%(uncovered.getOutOfGridXsec()) )
    print( "Total cross section in long cascade decays (fb): %10.3E\n" %(uncovered.
↪getLongCascadeXsec()) )
    print( "Total cross section in decays with asymmetric branches (fb): %10.3E\n"
↪%(uncovered.getAsymmetricXsec()) )

    #Print some of the missing topologies:
    print( 'Missing topologies (up to 3):' )
    for topo in uncovered.missingTopos.topos[:3]:
        print( 'Topology:',topo.topo )
        print( 'Contributing elements (up to 2):' )
        for el in topo.contributingElements[:2]:
            print( el,'cross-section (fb):', el.missingX )

    #Print elements with long cascade decay:
    print( '\nElements outside the grid (up to 2):' )
    for topo in uncovered.outsideGrid.topos[:2]:
        print( 'Topology:',topo.topo )
        print( 'Contributing elements (up to 4):' )
        for el in topo.contributingElements[:4]:
            print( el,'cross-section (fb):', el.missingX )
            print( '\tmass:',el.getMasses() )
```

output:

```
Total missing topology cross section (fb):  3.717E+03

Total cross section where we are outside the mass grid (fb):  9.984E+01

Total cross section in long cascade decays (fb):  8.992E+02

Total cross section in decays with asymmetric branches (fb):  2.790E+03

Missing topologies (up to 3):
Topology: [[],[]](MET,MET)
Contributing elements (up to 2):
[[],[]] cross-section (fb): 0.45069449699999997

Elements outside the grid (up to 2):
Topology: [[[W]],[[Z]]](MET,MET)
Contributing elements (up to 4):
[[[W+]],[[Z]]] cross-section (fb): 0.3637955190752928
        mass: [[2.93E+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 *final states* (or Z_2-even particles such as the Higgs, W,...) 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 .

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] [-k] [-6] [-8] [-n] [-N] [-O] -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.

-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

-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

-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.

- **The cross section calculation is implemented by the** computeXSec 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/slha/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] [-s SIGMACUT] [-illegal] [-dB] -f FILENAME

***arguments*:**

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

-xS, --xsec turn off the check for xsection blocks

-s SIGMACUT, --sigmacut SIGMACUT give sigmacut in fb

-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 -s 0.01 -f inputFiles/slha/lightSquarks.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 charged particles in the input file. Since the inclusion of HSCP signatures in SModelS, these checks are no longer done by the SLHA file checker. However, if the input model contains a considerable fraction of cross-section going into displaced vertex signatures, a warning is issued on the screen when running SModelS.

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-PAS-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
```

(continues on next page)

(continued from previous page)

```
In [5]: browser.getULFor('CMS-PAS-SUS-15-002','T1tttt',[[gluinoMass,LSPmass],
↪[gluinoMass,LSPmass]]) #Get UL for a specific experimental result
Out[5]: 5.03E-02 [pb]

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-15-002'] UL =  5.03E-02 [pb]
   ['CMS-PAS-SUS-16-014'] UL =  4.10E-02 [pb]
   ['CMS-PAS-SUS-16-015'] UL =  1.80E-02 [pb]
   ['CMS-PAS-SUS-16-016'] UL =  5.76E-02 [pb]


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-15-002'] [2.20E+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]]
```

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] -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]

-f SMODELSFOLDER, --smodelsFolder SMODELSFOLDER path to the smodels folder with the SModelS python output files.

-s SLHAFOLDER, --slhaFolder SLHAFOLDER path to the SLHA folder 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 3x9 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 inteactive 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. 'm_gluino': ['MASS', 1000021].
  - **variable_x**: In a dictionary form, give the name of the x-axis variable, and the block and PDG code number to find it in the SLHA file. Example: variable_x = {'m_gluino[GeV]': ['MASS', 1000021]}.
  - **variable_y**: same for the y-axis. Example: variable_y = {'m_suR[GeV]': ['MASS', 2000002]}
- *spectrum hover information*: defines which information from the input SLHA file will appear in the hover box. The syntax is again a python dictonary.
  - **slha_hover_information**: information from the input SLHA file, e.g. model parameters or masses. Example: 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 = {'ctau_chi1+': 1000024}
  - **BR_hover_information**: defines for which particle(s) to display decay channels and branching ratios. Example: BR_hover_information = {'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 **BR_get_top**, e.g. BR_get_top = 5 (default: BR_get_top = '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', '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_long_xsec**: shows the total missing cross section in long cascade decays
- **MT_asym_xsec**: shows the total missing cross section in decays with asymmetric branches
- **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', 'chi2', 'Tx', 'Analysis', 'MT_max', 'MT_max_xsec', 'MT_total_xsec', 'MT_long_xsec', 'MT_asym_xsec']

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*.

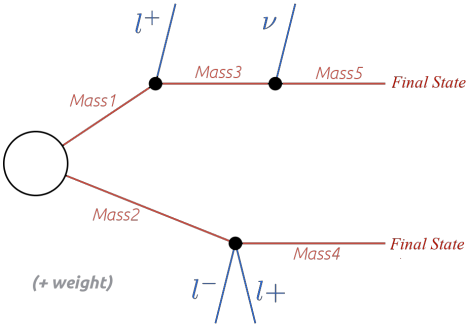
Simplified Model Definitions

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

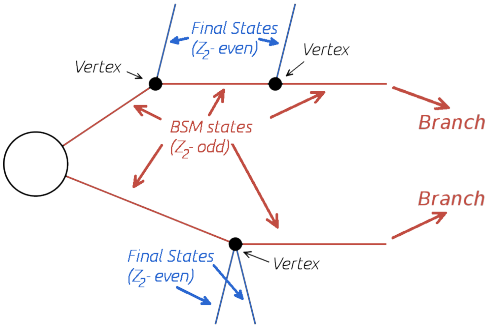
The applicability of SModelS is currently restricted to models which contain a Z_2 symmetry (R-parity in SUSY, K-parity in UED, ...). This is required in order to provide a clear structure for the simplified model topologies appearing during the *decomposition* of the input model. Below we describe the 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 masses of the BSM (Z_2-odd) states which have decayed or appear in the last step of the decay. Furthermore, the last BSM (Z_2-odd) particle is classified according to its quantum numbers as a specific *final state* class: *MET*, *HSCP*, *R-hadron*,etc. 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. For instance, as shown in the *scheme above*, only the masses of the BSM states and the quantum

[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.

number of the last BSM state are used, while other properties, such as their spins are ignored (the PID's are, however, stored for book-keeping).

Below we describe in more detail the element properties and their implementation in SModelS.

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

Z_2-even Final States

Z_2-even final states coming out of a vertex (see *scheme above*) usually correspond to Standard Model particles (electrons, gauge bosons, Higgs,...). However, if the input model contains Z_2-even BSM states (such as additional Higgs bosons), these also appear as final states. In contrast, stable or long-lived Z_2-odd particles which might appear in the detector (either as MET or charged tracks) are *not* classified as final states[2] .

- Z_2-even **states are defined in** smodels/share/default_particles.py

Z_2-odd Intermediate States

The intermediate Z_2-odd states are always assumed to consist of BSM particles with prompt Z_2 conserving decays of the form: (Z_2-odd state) $\rightarrow$ (Z_2-odd state') + *final states*. The only information kept from the intermediate states are their masses (see *scheme above*).

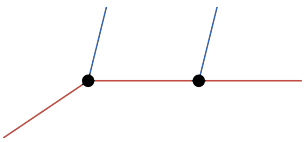
- Z_2-odd **states are defined by the input model file** (see *model* in *parameters file*)

Z_2-odd Final State Class

Besides the intermediate Z_2-odd BSM states, due to the assumed Z_2 symmetry, the element must also contain one stable Z_2-odd final state (at least in collider scales). The quantum numbers of this BSM final state are essential for defining which type of signature this element represents. In an element the Z_2-odd final state quantum numbers are mapped to a final state class, as defined in the particleNames module. Some examples of final state classes are: 'MET', 'HSCP' and 'RHadronQ'. Displaced decays, which currently cannot be tested in SModelS, are tracked with a proxy final state class 'Displaced', and will appear only in missing topologies. New final state classes can also be easily defined in this module.

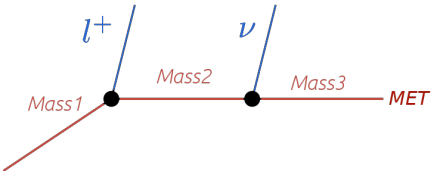
Branches

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.



[2] In order to shorten the notation we sometimes refer to Z_2-even final states simply as ''final states''. This should not be confused with the Z_2-odd *final state class*.

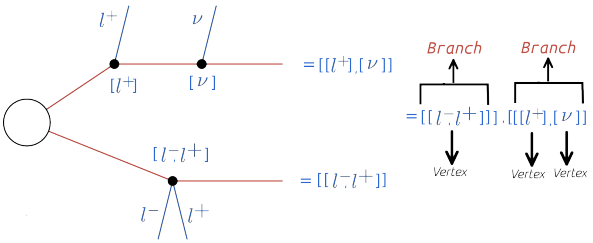
The structure of each branch is fully defined by its number of vertices and the number of *final states* coming out of each vertex. Furthermore, the branch also holds the information about the particle labels for the Z_2-even *final states* coming out of each vertex, the masses of the Z_2-odd states and the Z_2-odd *final state class* (e.g. 'MET'), as shown below.



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

Element Representation: Bracket Notation

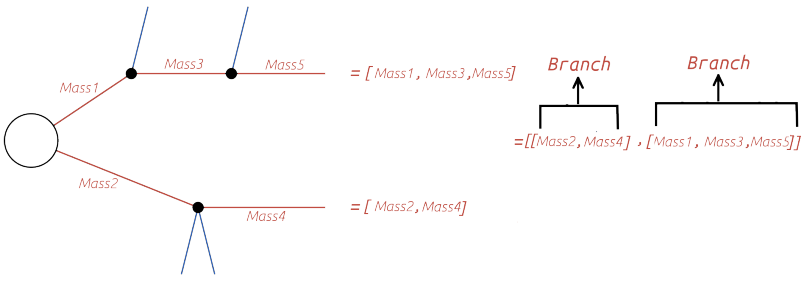
The structure and final states of *elements* are represented in textual form using a nested brackets notation. The scheme below shows how to convert between the graphical and bracket representations of an element:



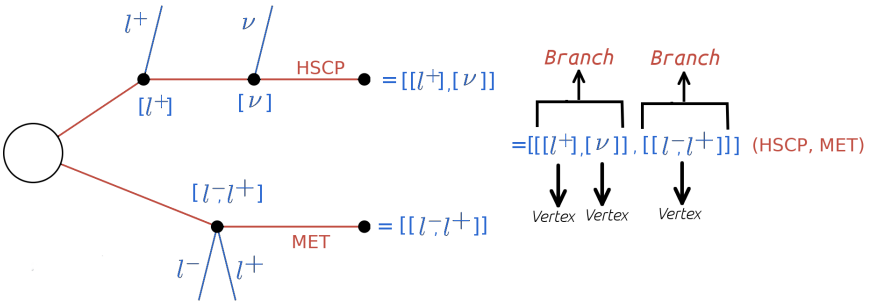
The brackets are ordered and nested in the following way. The outermost brackets correspond to the *branches* of the *element*. The branches are sorted according to their size (see *element sorting*) and each branch contains an *ordered* list of *vertices*. Each vertex contains a list of the Z_2-even *final states* (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]
```

Using the above scheme it is possible to unambiguously describe each *element* with a simple list of nested brackets. However, in order to fully specify all the information relative to a single *element*, we must also include the list of masses for the Z_2-odd states, the list of Z_2-odd *final state classes* and the element weight. The masses for the BSM (Z_2-odd) states can also be represented by a mass array for each branch, as shown below:



Finally the Z_2-odd *final state classes* can also be represented as a list in addition to the bracket notation:

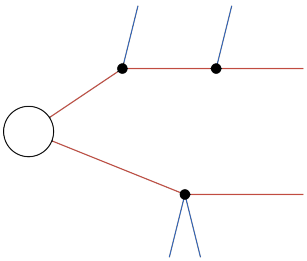


Topologies

It is often useful to classify *elements* according to their overall structure or topology. Each topology corresponds to an *undressed element*, removed of its Z_2-even *final states*, Z_2-odd final state class and Z_2-odd masses. Therefore the topology is fully determined by its number of branches, number of vertices in each *branch* and number of

Z_2-even *final states* 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 SUSY 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*.**

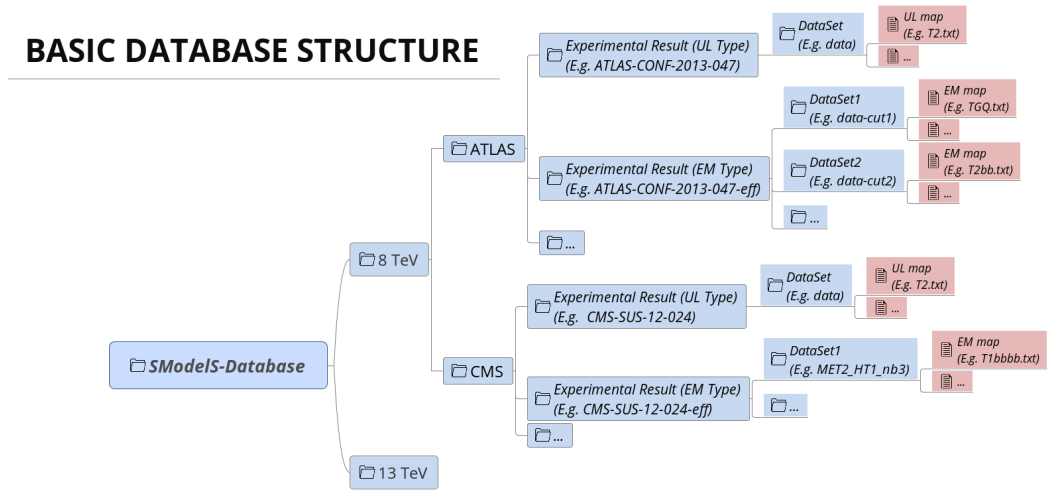
– *Experimental Result*: each *Experimental Result* corresponds to an experimental preliminary result (i.e. a CONF-NOTE or PAS) or publication and contains a list of *DataSets* as well as general information about the result (luminosity, publication reference,...).

  * *DataSet*: a single *DataSet* corresponds to one signal region of the experimental note or publication*[0]. In case of *UL-type results* there is a single *DataSet*, usually corresponding to the best signal region (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 refers to a single simplified model (or more precisely to a single *element* or sum of *elements*).

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

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

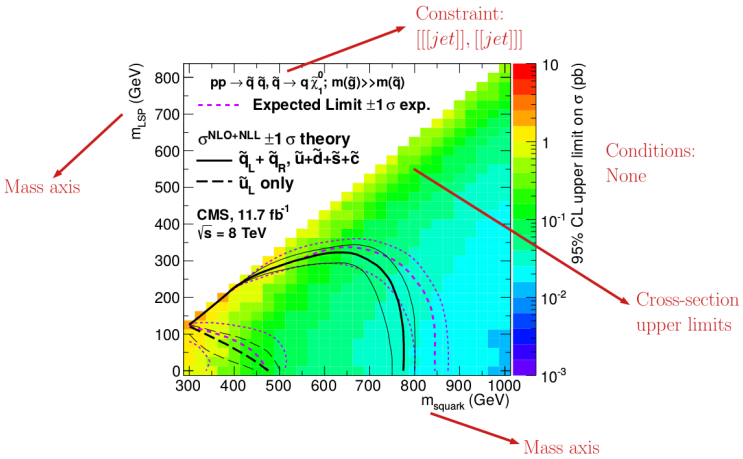
An experimental result contains all the relevant information corresponding to an experimental publication or preliminary result. In particular it holds general information about the experimental analysis, such as the corresponding luminosity, center of mass energy, publication reference, etc. The current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

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

[0] 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 Result: Upper Limit Type

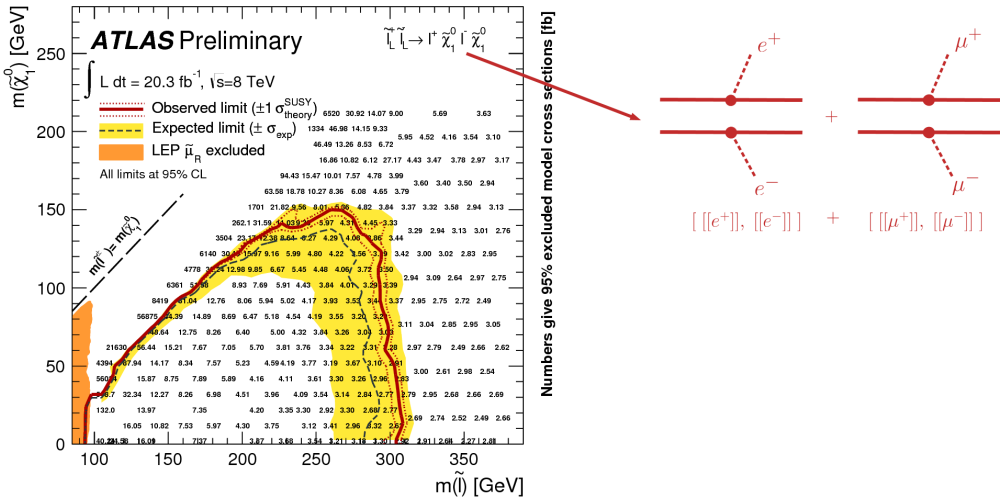
Upper Limit (UL) experimental results contain the experimental constraints on the cross section times branching ratio ($\sigma \times BR$) for Simplified Models from a specific experimental publication or preliminary result. These constraints are typically given in the format of Upper Limit maps, which 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 usually assume the best signal region (for a given point in parameter space), a combination of signal regions or more involved limits from other methods. 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 $\tilde{q} + \tilde{q} \to \left(jet + \tilde{\chi}_1^0\right) + \left(jet + \tilde{\chi}_1^0\right)$. Using the SModelS notation this simplified model is mapped to the *element* $[[[jet]], [[jet]]]$, using the notation defined in *Bracket Notation*. In addition to the constraint information, it is also possible to specificy a final state property for the simplified model, which corresponds to the BSM final state signature (see *Final State class*). If no final state is defined, the *element* is assumed to have a (MET, MET) final state signature. 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 is 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 constributions 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:

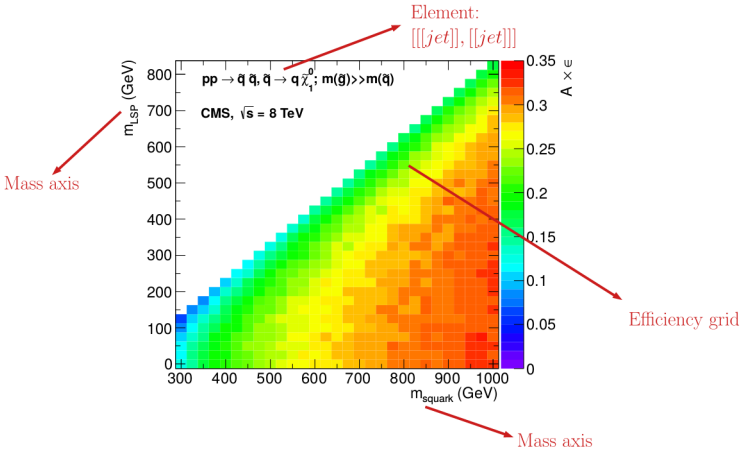
$$[[[e^+]], [[e^-]]] \simeq [[[\mu^+]], [[\mu^-]]] \quad \text{(fuzzy condition)}$$

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.

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 a 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 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*. Furthermore, a final state property can also be defined (see *Final State class*). If no final state is given, the *element* is assumed to have a (MET, MET) final state signature. 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 us to treat *EM-type results* and *UL-type results* in a very similar fashion (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

The main input for SModelS is the full model definition, which can be given in the two following forms:

- a SLHA (SUSY Les Houches Accord) file containing masses, branching ratios and cross sections for the BSM states (see an example file `here`)
- a 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, a LHE file can always be generated for any BSM model (through the use of your favorite MC generator).*[0] In this case, however, the precision of the results is limited to the MC statistics used to generate the file. *We also point out that all the decays appearing in the LHE input are assumed to be prompt and this input format should no be used if the model contains meta-stable particles*.

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 (see definition in *final states*) should be set as stable, since in SModelS they are effectively considered as final states.

[0] 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.

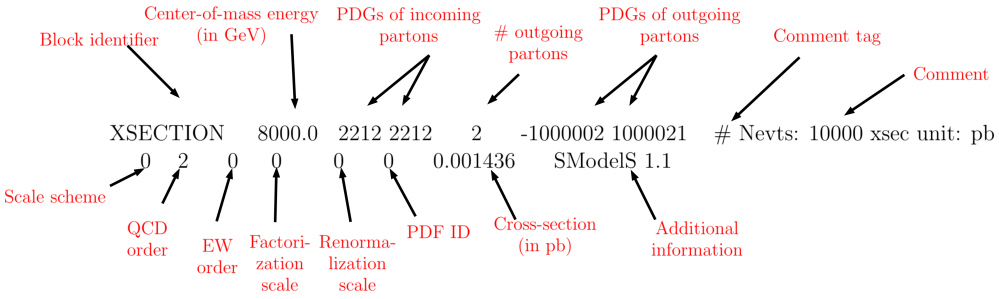
When generating the events it is also important to ensure that no mass smearing is applied, so the mass values for a given particle are the same throughout the LHE file.

New Particles

Besides information about the masses and branching ratios, the user must also define which particles are Z_2-odd states (*Intermediate states*) and which are Z_2-even (*Final states*). These definitions must be given in a model file, including the particle's quantum numbers, as illustrated in the `mssm.py` file. A path to a user's own model file can be specified in the *parameter file*, in the *[particles]* section.

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 $pp \to \tilde{\tau}_1^- + \tilde{\nu}_\tau$ 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**

Decomposition into Simplified Models

Given an input model, the first task of SModelS is to decompose the full model into a sum of Simplified Models (or *elements* in SModelS language). Based on the input format, which can be

- a SLHA file or
- a LHE file

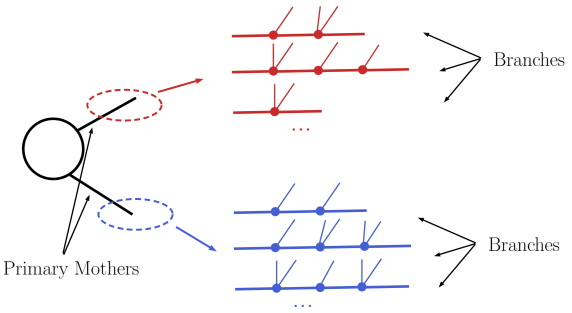
(see *Basic Input*), two distinct (but similar) decomposition methods are applied: the *SLHA-based* or the *LHE-based* decomposition.

SLHA-based Decomposition

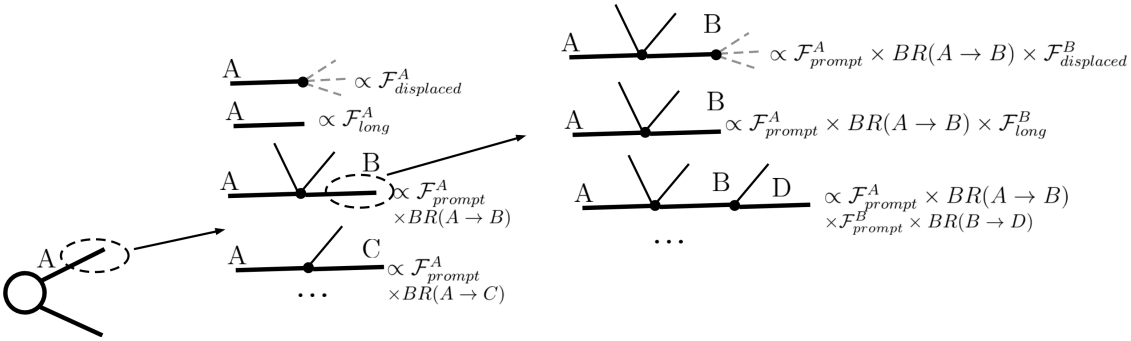
The SLHA file describing the input model is required to contain the masses of all the BSM states as well as their production cross sections, decay widths and branching ratios. All the above information must follow the guidelines of the SLHA format. In particular, the cross sections also have to be included as SLHA blocks according to the *SLHA cross section format*.

Once the production cross sections are read from the input file, all the cross sections for *production of two* Z_2-odd *states* are stored and 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 information contained in the DECAY blocks. This procedure is represented in the figure below:



Within SModelS all BSM particles are assumed to either decay promptly or to be stable (in detector scales). To deal with BSM particles with small (non-zero) width SModelS computes the probability for prompt decay ($\mathcal{F}_{prompt}$) as well as the probability for the particle to decay *outside* the detector ($\mathcal{F}_{long}$). If the particle has a considerable fraction of its decays taking place inside the detector (i.e $\mathcal{F}_{long} + \mathcal{F}_{prompt} \ll 1$), we include a third possibility, which is simply labeled as a *displaced decay*, with the fraction given by $\mathcal{F}_{displaced} = 1 - \mathcal{F}_{prompt} - \mathcal{F}_{long}$.[1] The branching fraction rescaled by $\mathcal{F}_{long}$ describes the probability of a decay where the daughter BSM state traverses the detector (thus is considered stable), while the branching fraction rescaled by $\mathcal{F}_{prompt}$ corresponds to a prompt decay which will be followed by the next step in the cascade decay. Finally, the rescaling by $\mathcal{F}_{displaced}$ corresponds to the fraction of displaced decays summed over all final states. This reweighting is illustrated in the figure below:



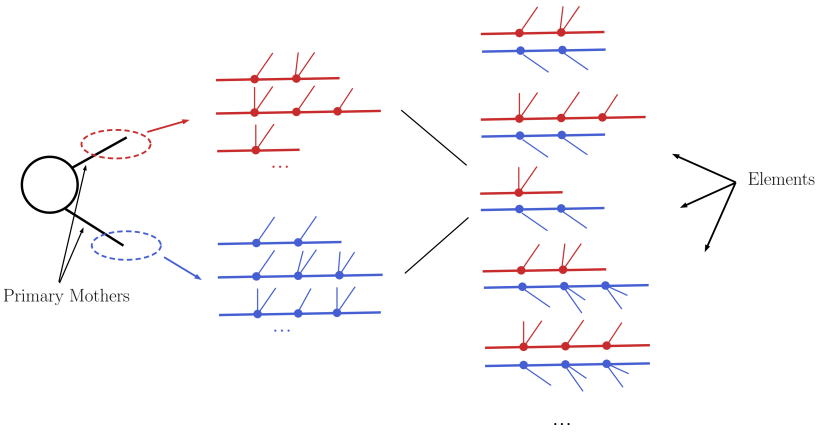
The precise values of $\mathcal{F}_{prompt}$ and $\mathcal{F}_{long}$ depend on the relevant detector size (L), particle mass (M), boost (β) and width (Γ), thus requiring a Monte Carlo simulation for each input model. Since this is not within the spirit of the simplified model approach, we approximate 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 effective size of the detector (which we take to be 10 m for both ATLAS and CMS), L_{inner} is the effective 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.

Following the above procedure it is possible to construct all cascade decay possibilities (including the stable case) for a given initial mother particle. Within the SModelS language each of the possible cascade decays 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:

[1] Note that the final states appearing in the displaced vertex are not stored during decomposition, since SModelS can not currently constrain displaced decay signatures. As a result, all elements with displaced decays will be identified as *missing topologies*.



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]

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 SLHA decomposition is a list of such topologies, where each topology contains a list of the elements generated during the decomposition.

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

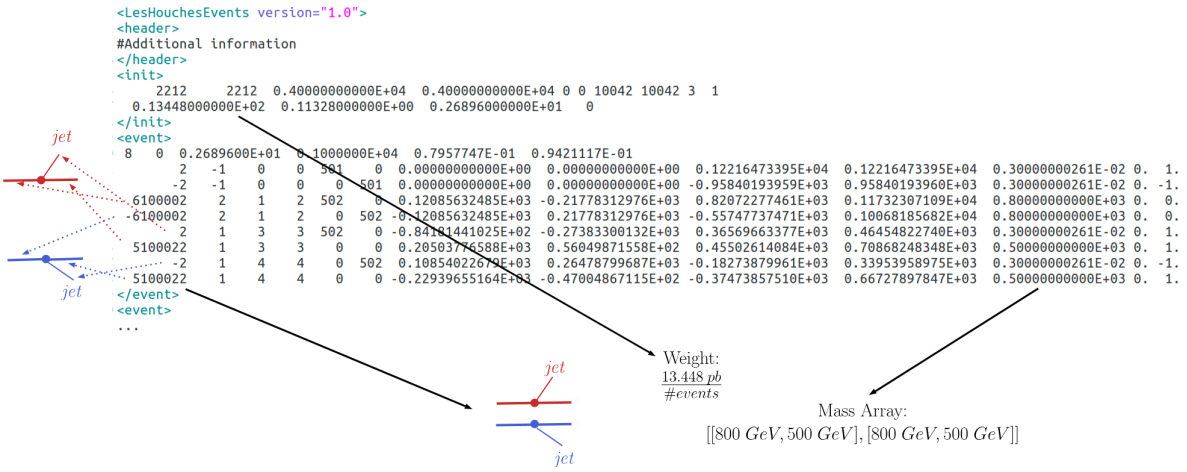
Minimum Decomposition Weight

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 times the width rescaling) can be discarded, since they will fall well below the experimental limits. Therefore, during the SLHA 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 given by the sigcut parameter and is easily adjustable (see slhaDecomposer.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 sigcut approximately one order of magnitude below the minimum signal cross sections the experimental data can constrain.

LHE-based Decomposition

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. Each event can then be directly mapped to an *element* with the element weight corresponding to the event weight. Finally, identical elements can be combined together (adding their weights). The procedure is represented in the example below:



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 should not be used for models with meta-stable BSM particles.

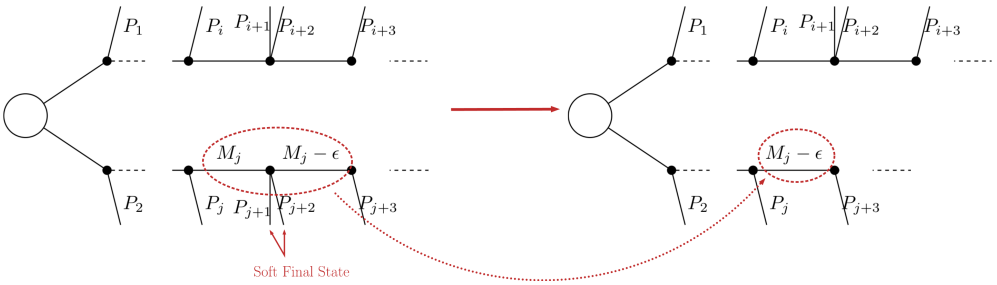
- **The LHE decomposition is implemented by the** LHE decompose method

Compression of Elements

During the decomposition process it is possible to perform several simplifications on the *elements* generated. In both the *LHE* and *SLHA*-based decompositions, 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 decay of an *intermediate state* 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:

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

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.

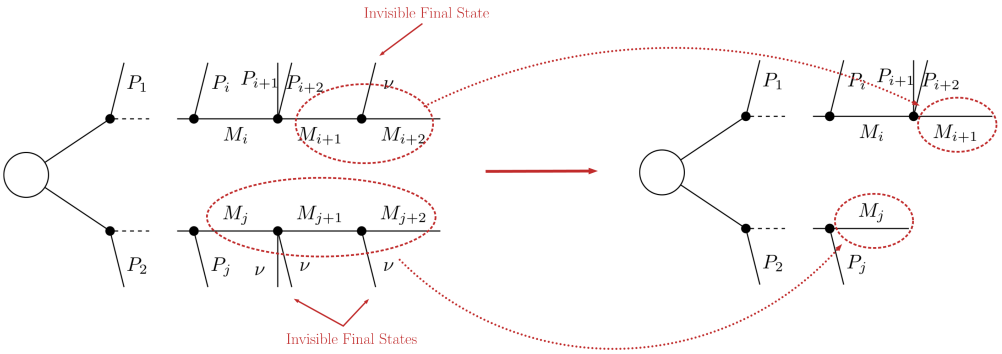
- **Mass compression is implemented by the** massCompress **method** and can be easily turned on/off by the flag *doCompress* in the *SLHA* or *LHE* decompositions.

Invisible Compression

Another type of compression is possible when the *final states* of the last decay are invisible. The most common example is

$$A \to \nu + B$$

as the last step of the decay chain, where B is an insivible particle leading to a MET signature (see *final state class*). 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, 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:



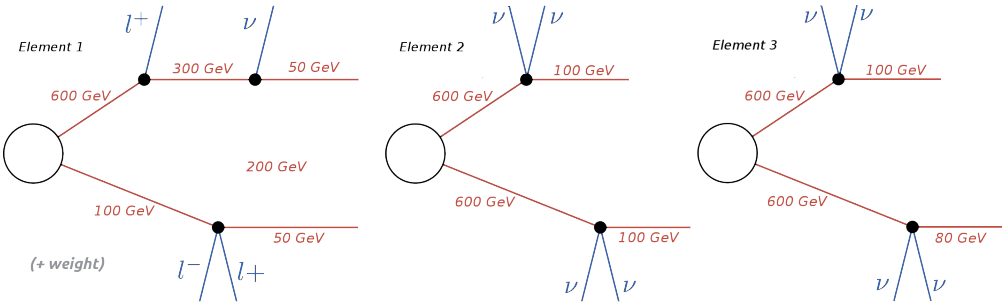
- **Invisible compression is implemented by the** invisibleCompress **method** and can be easily turned on/off by the flag *doInvisible* in the *SLHA* or *LHE* decompositions.

Element Sorting

In order to improve the code performance, *elements* created during *decomposition* and sharing a commong *topology* are sorted. Sorting allows for an easy 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 properties:

- Number of *vertices*
- Number of *final states* in each vertex
- *Final state* (Z_2-even) particles (particles belonging to the same vertex are alphabetically sorted)
- *Mass array*
- *Final state signature*

As an example, consider the three elements below:



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 mass array of Element 3 is smaller than the one 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

$$branch1 < branch2.$$

- **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 of one *element* or sum of *elements*. The *element* weight is defined as $\sigma \times BR \times \mathcal{F}$, where σ is the total production cross-section, BR is the product of all branching ratios for the decays appearing in the element and $\mathcal{F}$ is the product of all the lifetime reweighting factors ($\mathcal{F}_{long}$ and $\mathcal{F}_{prompt}$), as discussed in the *SLHA decomposition* (for the *LHE-type decomposition* $\mathcal{F} = 1$).

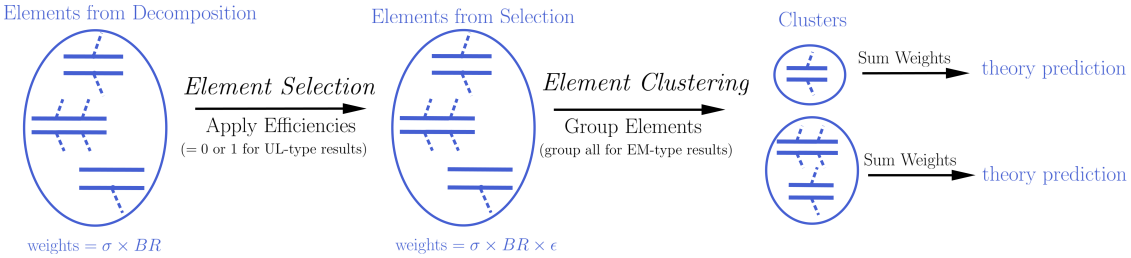
Therefore, in order to apply the experimental constraints, SModelS must first compute the theoretical value of $\sigma \times BR \times \mathcal{F}$ summing only over the *elements* appearing in the respective *constraint*. This is done applying a 1 (zero) efficiency (ϵ) for the elements which appear (do not appear) in the *constraint*. Then the final theoretical prediction is the sum over all *elements* with a non-zero value of $\sum \sigma \times BR \times \mathcal{F} \times \epsilon$. 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 ($\sum \sigma \times BR \times \mathcal{F} \times \epsilon$) in a given signal region (*DataSet*). Consequently, in this case SModelS must compute $\sigma \times BR \times \mathcal{F} \times \epsilon$ 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

$\sigma \times BR \times \mathcal{F} \times \epsilon$. 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 $\sigma \times BR \times \mathcal{F} \times \epsilon$ 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 type 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 ($\sigma \times BR \times \mathcal{F} \times \epsilon$) belonging to the same cluster:

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

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**

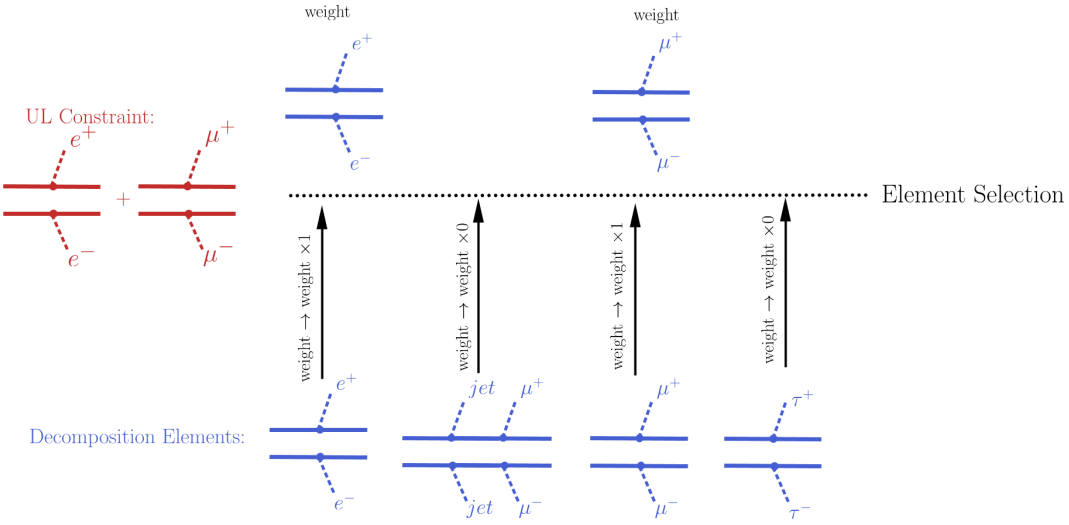
Theory Predictions for Upper Limit Results

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



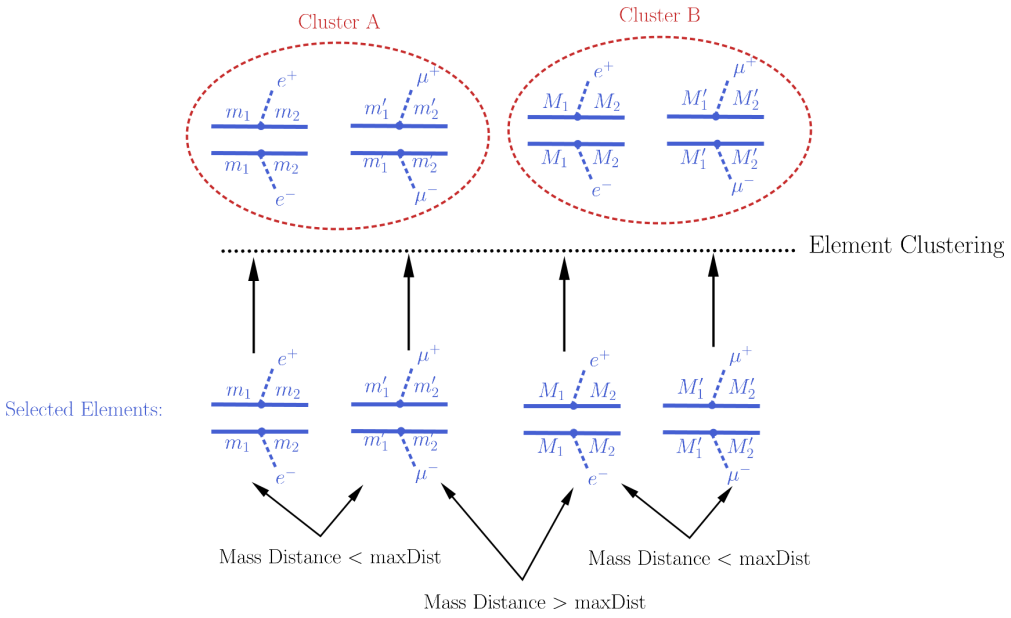
- **The element selection is implemented by the** getElementsFrom **method**

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 ($\sigma \times BR \times \mathcal{F}$) of all the selected *elements*. However, the selected *elements* usually differ in their masses*[0] and the experimental limit (see *Upper Limit constraint*) assumes that all the *elements* appearing in the *constraint* have the same mass (or mass array). As a result, the selected *elements* must be grouped into *clusters* of equal masses. When grouping the *elements*, however, one must allow for small mass differences, since the experimental efficiencies should not be strongly sensitive to small mass differences. 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 identical masses 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 masses” and the definition should depend on the *Experimental Result*, since different results will be more or less sensitive to mass differences. SModelS uses an UL map-dependent measure of the distance between two *element* masses, as described in *Mass Distance*.

If two of the selected *elements* have a *mass distance* smaller than a maximum value (defined by maxDist), they are gouped in the same mass cluster, as illustrated by the example below:

[0] As discussed in *Database Definitions*, *UL-type results* have a single *DataSet*.



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**.

Mass Distance

As mentioned *above*, in order to cluster the *elements* it is necessary to determine whether two *elements* have similar masses (see *element* and *Bracket Notation* for more details on *element* mass). Since an absolute definition of ''similar masses" is not possible and the sensitivity to mass differences depends on the experimental result, SModelS uses an ''upper limit map-dependent" definition. For each *element*'s mass array, the upper limit for the corresponding mass values is obtained from the UL map (see *UL-type result*). This way, each mass array is mapped to a single number (the cross section upper limit for the experimental result). Then the distance between the two *element*'s masses is simply given by the relative difference between their respective upper limits. More explicitly:

$$\begin{aligned} &\text{Element } A\ (M_A = [[M1, M2, ...], [m1, m2, ...]]) \rightarrow \text{ Upper Limit}(M_A) = x \\ &\text{Element } B\ (M_B = [[M1', M2', ...], [m1', m2', ...]]) \rightarrow \text{ Upper Limit}(M_B) = y \\ &\Rightarrow \text{mass distance}(A, B) = \frac{|x - y|}{(x + y)/2} \end{aligned}$$

where M_A, M_B (x, y) are the mass arrays (upper limits) for the *elements* A and B, respectively. If the mass distance of two *elements* is smaller than maxDist, the two masses are considered similar.

Notice that the above definition of mass distance quantifies the experimental analysis sensitivity to mass differences, which is the relevant parameter when *clustering elements*. Also, a check is performed to ensure that masses with very distinct values but similar upper limits are not clustered together.

- **The mass distance function is implemented by the** distance **method**

Theory Predictions for Efficiency Map Results

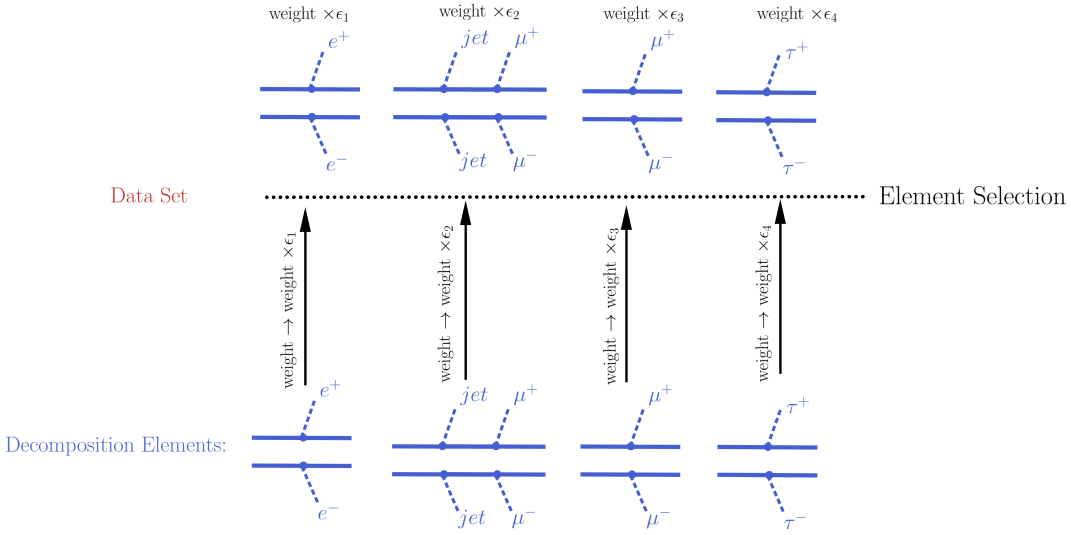
In order to compute the signal cross sections for a given *EM-type result*, so it can be compared to the signal region limits, it is first necessary to apply the efficiencies (see *EM-type result*) to all the *elements* generated by the model *decomposition*. Notice that typically a single *EM-type result* 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 an *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 a *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* mass and on its topology and 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†[0]. In the figure below we illustrate the element selection for the case of a *EM-type result*/*DataSet*:



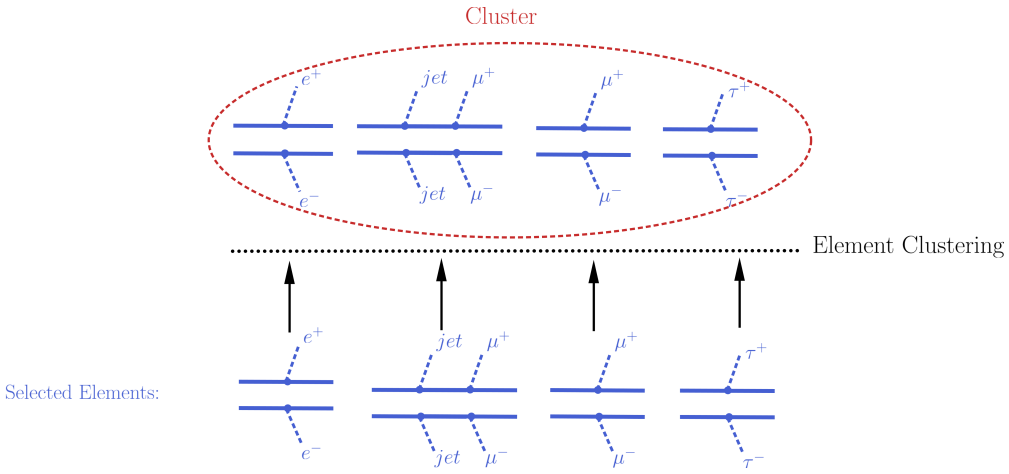
If, for instance, the analysis being considered vetoes $jets$ and τ's in the final state, we will have ϵ_2, $\epsilon_4 \simeq 0$ for the example in the *figure above*. Nonetheless, the element selection for a *DataSet* is usually more inclusive than the one applied for the *UL-type result*, resulting in less conservative values for the theory prediction.

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

Element Clustering

Unlike the clustering required in the case of *UL-type result* (see *Element Clustering for an UL analysis*), after the efficiencies have been applied to the element's weights, there is no longer the necessity to group the *elements* according to their masses, since the mass differences have already been accounted for by the different efficiencies. As a result, after the *element selection* all elements belong to a single cluster:

[0] When refering to an *element* mass, we mean all the *intermediate state* masses appearing in the *element* (or the *element* mass array). Two *elements* are considered to have identical masses if their mass arrays are identical (see *element* and *Bracket Notation* for more details).



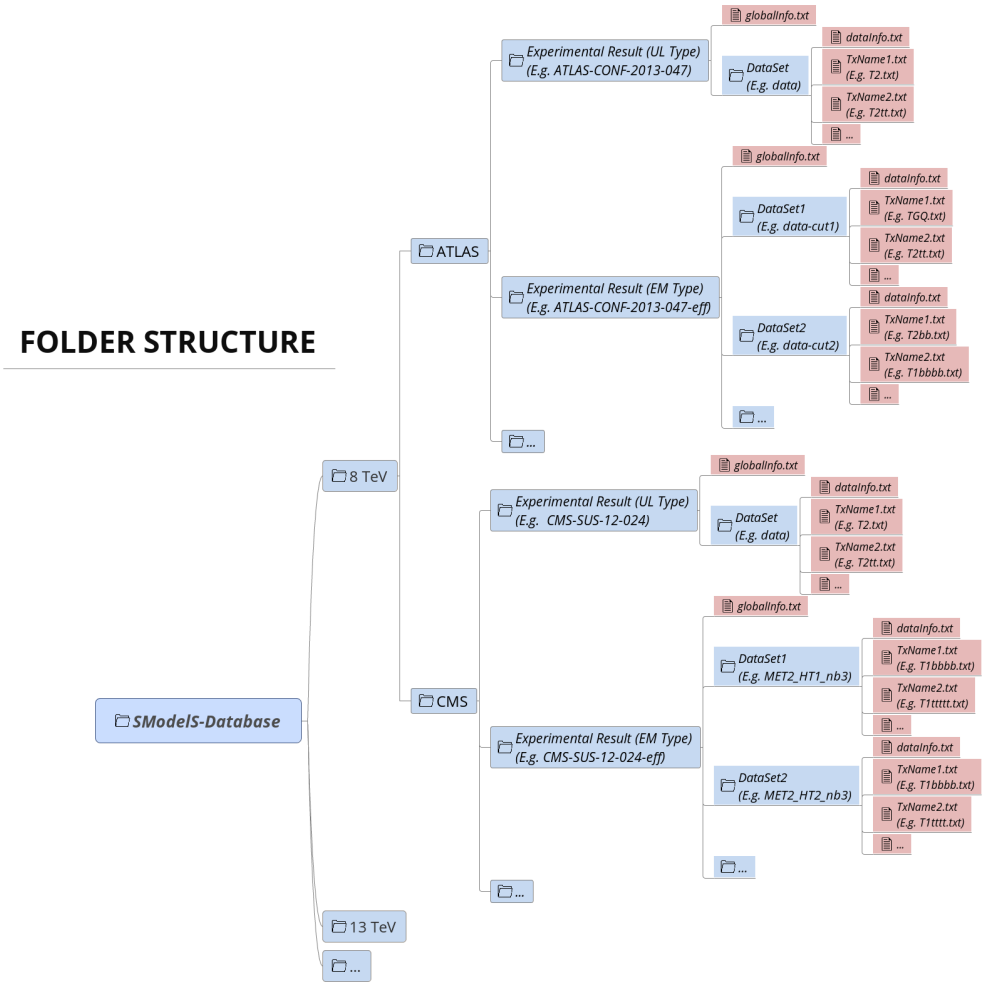
- **The (trivial) clustering of elements is implemented by the** clusterElements **method**.

Database of Experimental Results

SModelS stores all the information about the experimental results in the *Database*. Below we describe both the *directory* and *object* structure of the *Database*.

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
```

For *UL-type results*, each `TxName.txt` file contains the UL map for a given simplified model (*element* or sum of *elements*) as well as some meta information, including the corresponding *constraint* and *conditions*. 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
```

(continues on next page)

(continued from previous page)

```
source: CMS
validated: True
axes: [[x, y], [x, y]]
```

If the finalState property is not provided, the simplified model is assumed to contain neutral BSM final states in each branch, leading to a MET signature. However, if this is not the case, the non-MET final states must be explicitly listed in the `TxName.txt` file (see *final state classes* 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']
```

The second block of data in the `TxName.txt` file contains the upper limits as a function of the BSM masses:

```
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 UL map is given as a Python array with the structure: [[masses, upper limit], [masses, upper limit], ...].

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
```

For *EM-type results*, each `TxName.txt` file contains the efficiency map for a given simplified model (*element* or sum of *elements*) as well as some meta information. Here is the first few lines of CMS-SUS-13-012-eff/3NJet6_1000HT1250_200MHT300/T2.txt:

```
txName: T2
conditionDescription: None
```

(continues on next page)

(continued from previous page)

```
condition: None
constraint: [[['jet']],[['jet']]]
figureUrl: https://twiki.cern.ch/twiki/pub/CMSPublic/PhysicsResultsSUS13012/Fig_7a.pdf
validated: True
axes: 2*Eq(mother,x)_Eq(lsp,y)
publishedData: False
```

As seen above, the first block of data in the `T2.txt` file contains information about the *element* ([[[jet]], [[jet]]]) in *bracket notation* for which the efficiencies refers to as well as reference to the original data source and some additional information. As in the Upper Limit case, the simplified model is assumed to contain neutral BSM final states (MET signature). For non-MET final states the finalState field must list the *final state signatures*. The second block of data contains the efficiencies as a function of the BSM masses:

```
efficiencyMap: [[[[312.5*GeV, 12.5*GeV], [312.5*GeV, 12.5*GeV]], 0.00109],
[[[312.5*GeV, 62.5*GeV], [312.5*GeV, 62.5*GeV]], 0.00118],
[[[312.5*GeV, 112.5*GeV], [312.5*GeV, 112.5*GeV]], 0.00073],
[[[312.5*GeV, 162.5*GeV], [312.5*GeV, 162.5*GeV]], 0.00044],
        ...
```

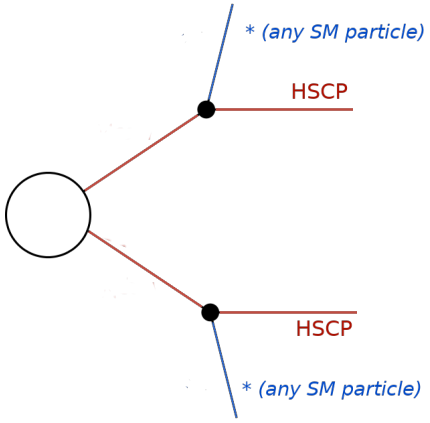
As we can see the efficiency map is given as a Python array with the structure: [[masses, efficiency], [masses, efficiency], ...].

Inclusive Simplified Models

If the analysis signal efficiencies 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: [[['*']],[['*']]]
```

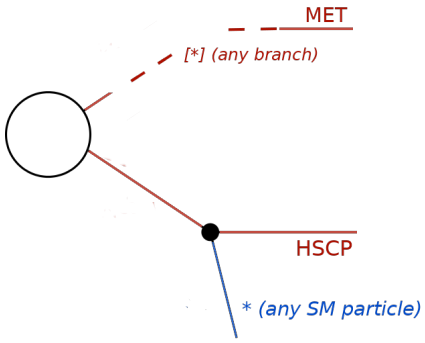
and represents the (inclusive) simplified model:



Note that although the final state represented by "*" is any Z_2-even *final states*, 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. Notice that the notation "[*]" corresponds to *any 'branch*, while ["*"] means *any particle*:

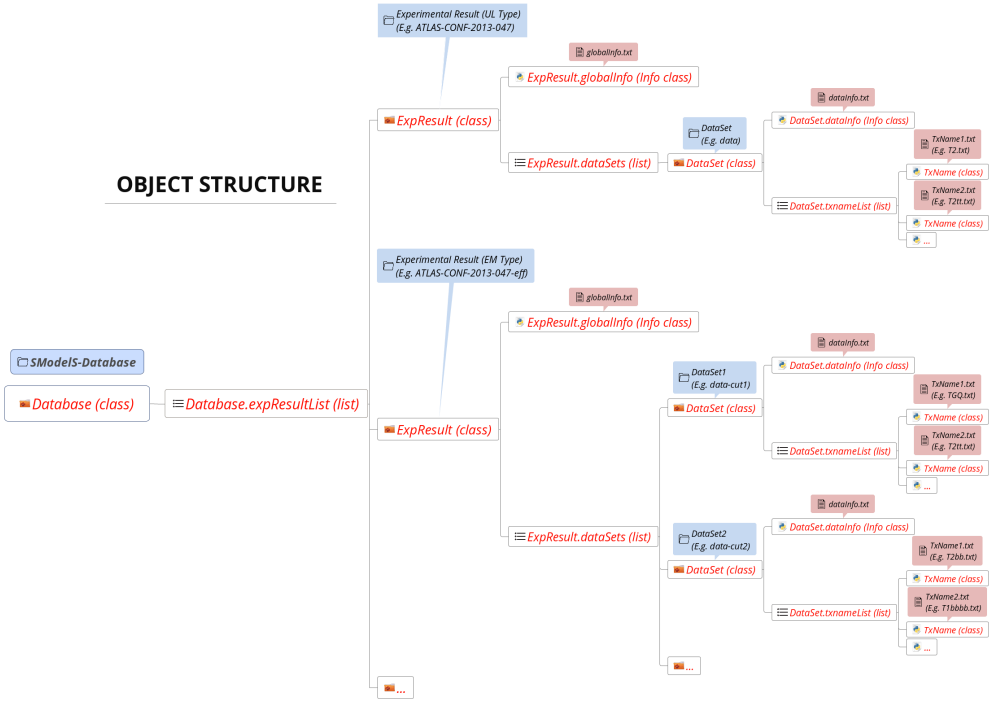


In such cases the mass array for the arbitrary branch must also be specified as using wildcards:

```
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, except for a few exceptions. Below we show the overall object structure as well as the folders/files the objects represent (click to enlarge):

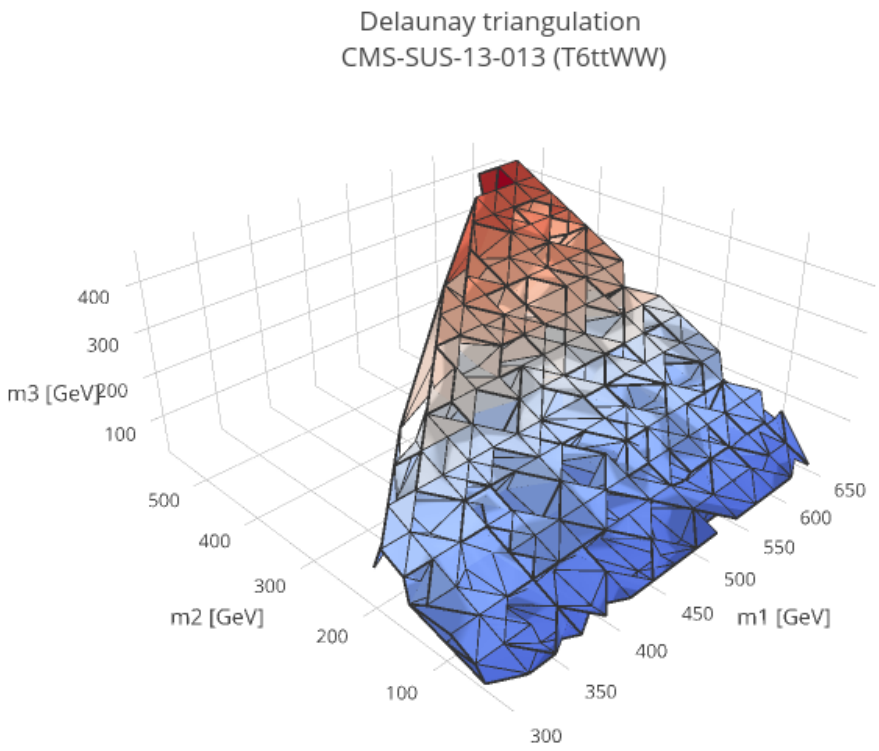


The type of Python object (Python class, Python list,...) is shown in brackets. For convenience, below we explicitly list the main database folders/files and the Python objects they are mapped to:

- *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 Figure below). The simplices defined during triangulation are then used for linearly interpolating the data grid, thus allowing SModelS to compute efficiencies or upper limits for arbitrary mass values (as long as they fall inside the data grid). 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.



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

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 (theory prediction)/(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. Following the procedure detailed in CMS-NOTE-2017-001, we construct a simplified likelihood which describes the plausibility of the data D, given a signal strength μ:

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

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

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

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. $T \gtrsim 1.96$ corresponds to a 95% confidence level upper limit. While n_{obs}, b and δ_b are directly extracted from the data set (coined *observedN*, *expectedBG* and *bgError*, respectively), n_{signal} is obtained from the calculation of the theory predictions. A default 20% systematical uncertainty is assumed for n_{signal}, resulting in $\delta^2 = \delta_b^2 + (0.2 n_{\text{signal}})^2$.

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

[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.

Combination of Signal Regions

If the experiment provides a covariance matrix together with the efficiency maps, signal regions can be combined. This 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.

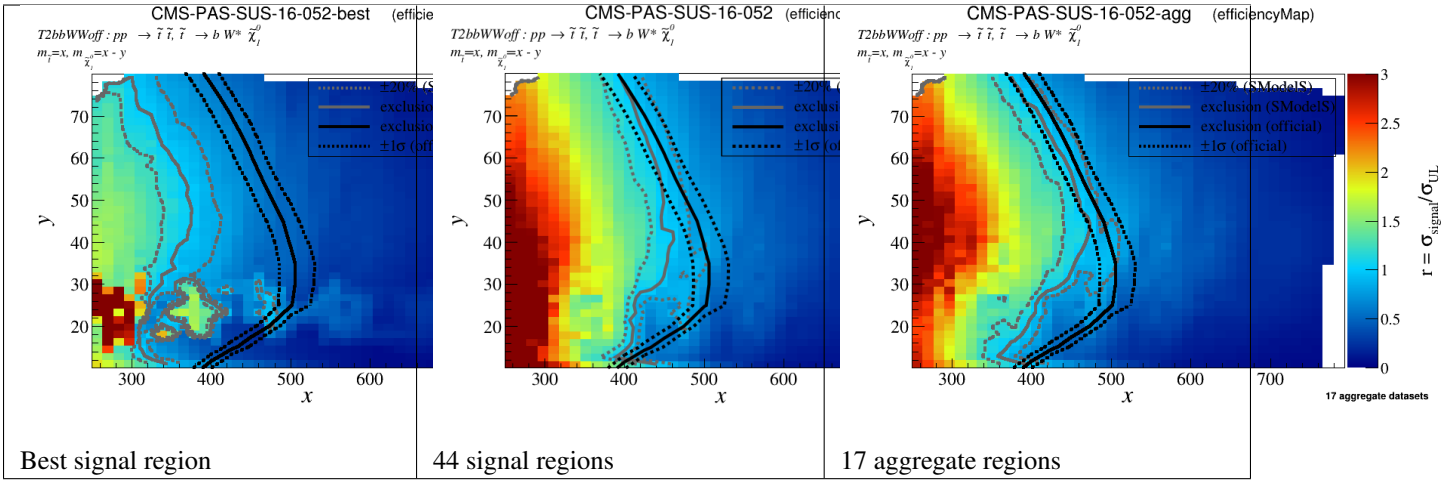


Figure: Comparison of exclusion curves for CMS-PAS-SUS-16-052 using only the best signal region (left), the combination of 17 aggregate signal regions (center), and the combination of all 44 signal regions (right).

Topology 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.

Given the *decomposition* output (list of *elements*), as well as the *database* information, it finds and classifies the *elements* which are not tested by any of the *experimental results* in the *database*. These elements are grouped into the following classes:

- *missingTopos*: *elements* which are not tested by any of the *experimental results* in the *database* (independent of the element mass). The missing topologies are further classified as:
  - *longCascade*: *elements* with long cascade decays (more than one intermediate particle in one of the *branches*);
  - *asymmetricBranches*: *elements* where the first *branch* differs from the second *branch* (but that are not considered as long cascade decays).
- *outsideGrid*: *elements* which could be tested by one or more experimental result, but are not constrained because the mass array is outside the mass grid;

In order to classify the *elements*, the tool loops over all the *elements* found in the *decomposition* and checks if they are tested by one or more *experimental results* in the *database**[0]. All the *elements* which are not tested by any of the *experimental results* in the *database* (independent of their masses) are added to the *missingTopos* class. The remaining *elements* which do appear in one or more of the *experimental results*, but have not been tested because their masses fall outside the efficiency or upper limit grids (see *EM-type results* and *UL-type results*), are added to the *outsideGrid* class.

Usually the list of *missing* or *outsideGrid* elements is considerably long. Hence, to compress this list, all *elements* differing only by their masses (with the same *final states*) or electric charges are combined. Moreover, by default, electrons and muons are combined to light leptons (denoted "l"): gluons and light quarks are combined into jets. The *missing* topologies are then further classified (if applicable) into *longCascade* or *asymmetricBranches* topologies.

[0] If *mass* or *invisible compression* are turned on, elements which can be *compressed* are not considered, to avoid double counting.

The topologies for each of the four categories are then grouped according to the final state (for the *missingTopos* and *outsideGrid* classes) or according to the PDG ids of the initially produced motherparticles (for the *longCascade* and *asymmetricBranches* classes). We note that for the latter the *elements* deriving from different mother particles, but with the same *final states* and mass configuration cannot be distinguished, and are therefore combined in this grouping. The full list of mother PDG id pairs can be accessed in the python printout or the comment of the text printout.

The topology coverage tool is normally called from within SModelS (e.g. when running *runSModelS.py*) by setting **testCoverage=True** in the *parameters file*. In the output, contributions in each category are ordered by cross section. By default only the ones with the ten largest cross sections are shown.

- **The topology coverage tool is implemented by the** Uncovered 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*[0].

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: 1.2.0
```

- 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* including the Z_2-odd *final state class*:

```
   ======================================================
 ||                                                    ||
 ||                  Selected Experimental Results             ||
 ||                                                    ||
   ======================================================
=======================================================
Experimental Result ID: ATLAS-SUSY-2015-01
Tx Labels: ['T2bb']
Sqrts: 1.30E+01 [TeV]
         ---------------------------
         Elements tested by analysis:
```

(continues on next page)

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

(continued from previous page)

```
            [[[b]],[[b]]] (MET, MET)
=======================================================
            [[[t,t]],[[t,t]]] (MET, MET)
=======================================================
Experimental Result ID: ATLAS-SUSY-2013-18
Tx Labels: ['T1bbbb', 'T1tttt']
Sqrts: 8.00E+00 [TeV]
         ----------------------------
         Elements tested by analysis:
            [[[b,b]],[[b,b]]] (MET, MET)
            [[[t,t]],[[t,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, *final states*, weight, the PIDs of the *intermediate particles* contributing to the *element* and the element ID:

```
   ======================================================
 ||                                                    ||
 ||                  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: ['MET', 'MET']
                The element masses are
                Branch 0: [1.29E+02 [GeV]]
                Branch 1: [1.29E+02 [GeV]]

                The element PIDs are
                PIDs: [[1000022], [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]
```

(continues on next page)

(continued from previous page)

```
Total Number of Elements: 7
                  ....................................................................
↪....
                  Element:
                  Element ID: 2
                  Particles in element: [[], [['W+']]]
                  Final states in element: ['MET', 'MET']
                  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], [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]

                  ....................................................................
↪....
                  Element:
                  Element ID: 3
                  Particles in element: [[], [['W-']]]
                  Final states in element: ['MET', 'MET']
                  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], [-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, 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: 1.5212733123965445
```

(continues on next page)

(continued from previous page)

```
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, -1000022], [2000004, 1000022]]
PIDs:[[-2000004, -1000022], [2000002, 1000022]]

---------------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: 1.35282642984693
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, -1000022], [2000004, 1000022]]
```

- summary information about the *missing topologies*, if **testCoverage** = True. The first information corresponds to the total input cross-section considered, which corresponds to the cross-section summed over all elements after decomposition. Note that this value might differ from the total input model cross-section, since it includes the *lifetime reweighting* and the effect of skipping elements with a weight below the *minimum decomposition weight*. The total missing topology cross section shown corresponds to the sum of cross sections of all *elements* which are not tested by any *experimental result*. If, however, the *element* is constrained by one or more *experimental results*, but its mass is 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. Finally, the *elements* which contribute to the total missing topology cross section are subdivided into *elements* with long decays or with asymmetric branches (see *coverage tool* for more details)

```
Total cross section considered (fb):  3.635E+03
Total missing topology cross section (fb):  3.037E+03
Total cross section where we are outside the mass grid (fb):  3.376E+00
Total cross section in long cascade decays (fb):  1.317E+03
Total cross section in decays with asymmetric branches (fb):  1.686E+03
```

- detailed information about the missing topologies with highest cross sections. The *element* cross section (weight) as well as its description in *bracket notation* and *BSM final state classification* are included. 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 [987, 988, 989, 990, 1001, 1002, 1003, 1004, 1033, 1034, 1035,␣
↪1036, 1047, 1048, 1049, 1050, 1061, 1062, 1063, 1064, 1075, 1076, 1077, 1078]
```

- 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.

```
================================================================================
Contributions outside the mass grid (up to 10):
Sqrts (TeV)   Weight (fb)        Element description
```

(continues on next page)

(continued from previous page)

```
 13.0  1.438E+00    #                [[[t],[W]],[[t],[W]]](MET,MET)
Contributing elements [813, 814, 815]
```

- information about the missing topologies with long cascade decays. The long cascade decays are classified by the initially produced mother particles. If more than one pair of mothers are contributing to the same class of elements, the full list is given in the comment. For definiteness all lists are sorted. If **addCoverageID** = True, all the *elements* IDs contributing to the missing topology are shown.

```
Mother1 Mother2 Weight (fb) # allMothers
1000021 2000002  3.696E+02 # [[1000021, 2000002]]
Contributing elements [1560, 1561, 1562, 1563, 1564, 1916, 1917, 1921, 1925, 1929,␣
↪1930, 1934, 1938, 1942, 1946, 1950, 1951, 1955, 1965, 1969, 1973, 1977, 1981, 1985,␣
↪1989, 1993, 1997, 2001, 2005, 2009, 2013, 2017, 2028, 2032, 2036, 2040, 2044, 2048,␣
↪2052, 2056, 2060, 2064, 2068, 2072, 2076, 2080, 2084, 2087, 2091, 2095, 2099, 2102,␣
↪2106, 2110, 2114, 2118, 2122, 2125, 2129, 2130, 3302, 3305, 3309, 3313, 3317, 3320,␣
↪3324, 3328, 3332, 3336, 3340, 3343, 3347, 3348, 3361, 3365, 3369, 3373, 3377, 3381,␣
↪3385, 3389, 3393, 3397, 3401, 3405, 3409, 3413, 3433, 3437, 3441, 3445, 3449, 3453,␣
↪3457, 3461, 3465, 3469, 3473, 3477, 3481, 3485, 3505, 3509, 3513, 3517, 3521, 3525,␣
↪3529, 3533, 3537, 3541, 3545, 3549, 3553, 3557, 3561, 3564, 3568, 3572, 3576, 3579,␣
↪3583, 3587, 3591, 3595, 3599, 3602, 3606, 3607, 3618, 3622, 3626, 3630, 3634, 3638,␣
↪3642, 3646, 3650, 3654, 3658, 3662, 3666, 3670, 3690, 3694, 3698, 3702, 3706, 3710,␣
↪3714, 3718, 3722, 3726, 3730, 3734, 3738, 3742, 3762, 3766, 3770, 3774, 3778, 3782,␣
↪3786, 3790, 3794, 3798, 3802, 3806, 3810, 3814, 3834, 3838, 3842, 3846, 3850, 3854,␣
↪3858, 3862, 3866, 3870, 3874, 3878, 3882, 3886, 3906, 3910, 3914, 3918, 3922, 3926,␣
↪3930, 3934, 3938, 3942, 3946, 3950, 3954, 3958, 3962, 3965, 3969, 3973, 3977, 3980,␣
↪3984, 3988, 3992, 3996, 4000, 4003, 4007, 4008, 4019, 4023, 4027, 4031, 4035, 4039,␣
↪4043, 4047, 4051, 4055, 4059, 4063, 4067, 4071, 4075, 4076, 4080, 4084, 4088, 4089,␣
↪4093, 4097, 4101, 4105, 4109, 4110, 4114, 4123, 4127, 4131, 4135, 4139, 4143, 4147,␣
↪4151, 4155, 4159, 4163, 4167, 4171, 4174]
1000021 2000001  2.031E+02 # [[1000021, 2000001], [1000021, 2000003]]
Contributing elements [1971, 1975, 1979, 1987, 1991, 1995, 1999, 2003, 2011, 2019,␣
↪2034, 2038, 2042, 2050, 2054, 2058, 2062, 2066, 2074, 2082, 3311, 3315, 3326, 3330,␣
↪3334, 3338, 3367, 3371, 3375, 3383, 3387, 3391, 3395, 3399, 3407, 3415, 3435, 3439,␣
↪3443, 3447, 3455, 3459, 3463, 3467, 3471, 3479, 3487, 3507, 3511, 3515, 3519, 3527,␣
↪3531, 3535, 3539, 3543, 3551, 3559, 3624, 3628, 3632, 3640, 3644, 3648, 3652, 3656,␣
↪3664, 3672, 3692, 3696, 3700, 3704, 3712, 3716, 3720, 3724, 3728, 3736, 3744, 3764,␣
↪3768, 3772, 3776, 3784, 3788, 3792, 3796, 3800, 3808, 3816, 3836, 3840, 3844, 3848,␣
↪3856, 3860, 3864, 3868, 3872, 3880, 3888, 3908, 3912, 3916, 3920, 3928, 3932, 3936,␣
↪3940, 3944, 3952, 3960, 4025, 4029, 4033, 4041, 4045, 4049, 4053, 4057, 4065, 4073,␣
↪4129, 4133, 4137, 4145, 4149, 4153, 4157, 4161, 4169, 1919, 1923, 1927, 1932, 1936,␣
↪1940, 1944, 1948, 1953, 1957, 1967, 1983, 2007, 2015, 2030, 2046, 2070, 2078, 2086,␣
↪2089, 2093, 2097, 2101, 2104, 2108, 2112, 2116, 2120, 2124, 2127, 2132, 3304, 3307,␣
↪3319, 3322, 3342, 3345, 3350, 3363, 3379, 3403, 3411, 3451, 3475, 3483, 3523, 3547,␣
↪3555, 3563, 3566, 3570, 3574, 3578, 3581, 3585, 3589, 3593, 3597, 3601, 3604, 3609,␣
↪3620, 3636, 3660, 3668, 3708, 3732, 3740, 3780, 3804, 3812, 3852, 3876, 3884, 3924,␣
↪3948, 3956, 3964, 3967, 3971, 3975, 3979, 3982, 3986, 3990, 3994, 3998, 4002, 4005,␣
↪4010, 4021, 4037, 4061, 4069, 4078, 4082, 4086, 4091, 4095, 4099, 4103, 4107, 4112,␣
↪4116, 4125, 4141, 4165, 4173, 4176]
```

- information about the missing topologies with asymmetric decays. The asymmetric branch decays are classified by the initially produced mother particles. If more than one pair of mothers are contributing to the same class of elements, the full list is given in the comment. For definiteness all lists are sorted. If **addCoverageID** = True, all the *elements* IDs contributing to the missing topology are shown.

```
Mother1 Mother2 Weight (fb) # allMothers
1000021 1000021  5.297E+02 # [[1000021, 1000021]]
```

(continues on next page)

(continued from previous page)

```
Contributing elements [565, 566, 567, 568, 569, 570, 571, 572, 573, 574, 575, 576,␣
↪577, 578, 579, 580, 581, 582, 583, 584, 585, 586, 587, 588, 589, 590, 591, 592, 593,
↪ 594, 595, 596, 597, 598, 599, 600, 601, 602, 603, 604, 605, 606, 607, 608, 609,␣
↪610, 611, 612, 613, 614, 615, 616, 617, 618, 619, 620, 1414, 1415, 1416, 1417, 1418,
↪ 1419, 1420, 1421, 1422, 1423, 1424, 1425, 1426, 1428, 1429, 1430, 1431, 1432, 1433,
↪ 1434, 1435, 1436, 1437, 1438, 1439, 1441, 1442, 1443, 1444, 1445, 1446, 1447, 1448,
↪ 1449, 1450, 1451, 1453, 1454, 1455, 1456, 1457, 1458, 1459, 1460, 1461, 1462, 1464,
↪ 1465, 1466, 1467, 1468, 1470, 1471, 1472, 1474, 1475, 1476, 1477, 1478, 1479, 1480,
↪ 1481, 1486, 1487, 1488, 1489, 1493, 1494, 1495, 1496, 1499, 1500, 1501, 1502, 1504,
↪ 1505, 1506, 1507, 1509, 1510, 1511, 1513, 1514, 1516]
1000002 1000021  4.149E+02 # [[1000002, 1000021], [1000004, 1000021]]
Contributing elements [466, 478, 480, 492, 494, 504, 506, 1027, 1028, 1029, 1030,␣
↪1031, 1032, 1033, 1034, 1035, 1036, 1037, 1038, 1039, 1040, 1055, 1056, 1057, 1058,␣
↪1059, 1060, 1061, 1062, 1063, 1064, 1065, 1066, 1068, 422, 423, 424, 425, 426, 427,␣
↪428, 429, 430, 431, 432, 433, 434, 435, 468, 470, 472, 482, 484, 496, 498, 508, 510,
↪ 514, 518, 523, 527, 531, 535, 539, 543, 547, 551, 556, 560, 1067, 1083, 1084, 1085,
↪ 1086, 1087, 1088, 1089, 1090, 1091, 1092, 1093, 1094, 1095, 1096, 1111, 1112, 1113,
↪ 1114, 1115, 1116, 1117, 1118, 1119, 1120, 1121, 1122, 1123, 1124, 1222, 1223, 1224,
↪ 1225, 1226, 1227, 1228, 1229, 1230, 1231, 1265, 1266, 1267, 1268, 1269, 1270, 1271,
↪ 1272, 1273, 1274, 1275, 1276, 1277, 1278, 1321, 1322, 1323, 1324, 1325, 1326, 1327,
↪ 1328, 1329, 1330, 1331, 1332, 1333, 1334, 1377, 1378, 1379, 1380, 1381, 1382, 1383,
↪ 1384, 1385, 1386, 1387, 1388, 1389, 477, 491, 1009, 1010, 1011, 1012, 1013, 1014,␣
↪1015, 1016, 1017, 1018, 1019, 1020, 1021, 1022, 1023, 1024, 1025, 1026]
```

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 = 0.01
# Database version: 1.2.0
```

- 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  1.169E+01  7.683E+00  1.521E+00  N/A
 Signal Region:  (UL)
 Txnames:  T2
--------------------------------------------------------------------------------
     CMS-SUS-16-033  1.30E+01    0.0  1.169E+01  8.639E+00  1.353E+00  N/A
 Signal Region:  (UL)
 Txnames:  T2
--------------------------------------------------------------------------------
 ATLAS-SUSY-2013-02  8.00E+00    0.0  6.270E-01  1.818E+00  3.449E-01  4.146E-01
 Signal Region:  SR2jt
 Txnames:  T1, T2
 Chi2, Likelihood =  1.302E-01  1.276E-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 = 1.5212733123965445
```

- summary information about the *missing topologies*, if **testCoverage** = True. The first information corresponds to the total input cross-section considered, which corresponds to the cross-section summed over all elements after decomposition. Note that this value might differ from the total input model cross-section, since it includes the *lifetime reweighting* and the effect of skipping elements with a weight below the *minimum decomposition weight*. The total missing topology cross section corresponds to the sum of all *elements* cross sections which are not tested by any *experimental result*. If, however, the *element* is constrained by one or more *experimental results*, but its mass is 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. Finally, the *elements* which contribute to the total missing topology cross section are subdivided into *elements* with long decays or with asymmetric branches (see *coverage tool* for more details)

```
Total cross section considered (fb):  3.635E+03
Total missing topology cross section (fb):  3.037E+03
Total cross section where we are outside the mass grid (fb):  3.376E+00
Total cross section in long cascade decays (fb):  1.317E+03
Total cross section in decays with asymmetric branches (fb):  1.686E+03
```

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

```
================================================================================
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  1.445E+02    # [[[jet,jet],[W]],[[jet],[jet,jet],[W]]](MET,MET)
```

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

```
================================================================================
Contributions outside the mass grid (up to 10):
Sqrts (TeV)   Weight (fb)        Element description
 13.0  1.438E+00    #              [[[t],[W]],[[t],[W]]](MET,MET)
 13.0  2.871E-01    #        [[[b],[higgs]],[[b],[higgs]]](MET,MET)
```

- information about the missing topologies with long cascade decays:

```
Missing topos: long cascade decays (up to 10 entries), sqrts = 13 TeV:
Mother1 Mother2 Weight (fb) # allMothers
1000021 2000002  3.696E+02 # [[1000021, 2000002]]
1000021 2000001  2.031E+02 # [[1000021, 2000001], [1000021, 2000003]]
1000002 1000021  1.612E+02 # [[1000002, 1000021]]
```

- information about the missing topologies with asymmetric decays:

```
Missing topos: asymmetric branches (w/o long cascades, up to 10), sqrts = 13 TeV
Mother1 Mother2 Weight (fb) # allMothers
1000021 1000021  5.297E+02 # [[1000021, 1000021]]
1000002 1000021  4.149E+02 # [[1000002, 1000021], [1000004, 1000021]]
1000021 2000002  2.134E+02 # [[1000021, 2000002]]
```

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 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': 0.01, 'minmassgap': 5.0, 'maxcond': 0.2, 'ncpus': 1,
↪'file status': 1, 'decomposition status': 1, 'warnings': 'Input fil
```

- 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': 0.480, 'xsec 13.
↪0 TeV': 1.580}, 'final states': ['MET', 'MET']},
{'ID': 2, 'Particles': "[[], [['W+']]]", 'Masses (GeV)': [[129.0], [269.0,129.0]],
↪'PIDs': [[[1000022], [1000024, 1000022]]], 'Weights (fb)': {'xsec 8.0 TeV': 0.2404,
↪'xsec 13.0 TeV': 0.263}, 'final states': ['MET', 'MET']},..]
```

- 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. The masses of the *elements* contributing to the signal cross section (if unique) and the χ^2 likelihood values (if **computeStatistics** = True) are also included. Finally, if **addTxWeights** = True, the weight contribution of each txname is also included:

```
'ExptRes': [{'maxcond': 0.0, 'theory prediction (fb)': 11.68725, 'upper limit (fb)':␣
↪7.68,
'expected upper limit (fb)': None, 'TxNames': ['T2'], 'Mass (GeV)': [[991.3084, 129.
↪0], [991.4, 129.0]],
'AnalysisID': 'CMS-SUS-16-036', 'DataSetID': None, 'AnalysisSqrts (TeV)': 13.0,
'lumi (fb-1)': 35.9, 'dataType': 'upperLimit',
'r': 1.52127, 'r_expected': None, 'TxNames weights (fb)': {'T2': 11.687}},...]
```

- a list of missing topologies (if **testCoverage** = True), stored under the *Missed Topologies* key. For each list entry, the *element* cross section (weight), the *element* IDs contributing to the topology and the *element* description in *bracket notation* and *final states* is included.

```
'Missed Topologies': [{'sqrts (TeV)': 13.0, 'weight (fb)': 148.16,
'element': "[[[jet],[W]],[[jet,jet],[W]]](MET,MET)", 'element IDs': [987, 988,...]},..
↪.]
```

- a list of topologies which are outside the *experimental results* grid (if **testCoverage** = True), stored under the *Outside Grid* key. For each list entry, the *element* cross section (weight) and the *element* description in *bracket notation* and *final states* is included.

```
'Outside Grid': [{'sqrts (TeV)': 13.0, 'weight (fb)': 1.58, 'element': "[[[jet]],[[t,
↪t]]](MET,MET)"},...]
```

- a list of topologies with long cascade decays (if **testCoverage** = True), stored under the *Long Cascades* key. For each list entry, the *element* cross section (weight) and the PIDs of the mothers are included. The mother PIDs are given in a nested list, as more than one pair might contribute to the same class of *elements*.

```
'Long Cascades': [{'sqrts (TeV)': 13.0, 'weight (fb)': 369.58, 'mother PIDs':␣
↪[[1000021, 2000002]]},...]
```

- a list of topologies with asymmetric branch decays (if **testCoverage** = True), stored under the *Asymmetric Branches* key. For each list entry, the *element* cross section (weight) and the PIDs of the mothers are included. The mother PIDs are given in a nested list, as more than one pair might contribute to the same class of *elements*.

```
'Asymmetric Branches': [{'sqrts (TeV)': 13.0, 'weight (fb)': 529.73, 'mother PIDs':␣
↪[[1000021, 1000021]]},...]
```

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:

```
<?xml version="1.0" ?>
<smodelsOutput>
    <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_List>
                        <PIDs>1000022</PIDs>
                    </PIDs_List>
```

(continues on next page)

(continued from previous page)


```
                    <PIDs_List>
                        <PIDs>1000022</PIDs>
                    </PIDs_List>
                </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>MET</final_states>
                <final_states>MET</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>991.3084550191742</Mass_GeV>
                    <Mass_GeV>129.0</Mass_GeV>
                </Mass_GeV_List>
                <Mass_GeV_List>
                    <Mass_GeV>991.4592631024188</Mass_GeV>
                    <Mass_GeV>129.0</Mass_GeV>
                </Mass_GeV_List>
            </Mass_GeV_List>
            <TxNames_List>
                <TxNames>T2</TxNames>
            </TxNames_List>
            <TxNames_weights_fb>
                <T2>11.687259769407873</T2>
            </TxNames_weights_fb>
            <dataType>upperLimit</dataType>
            <expected_upper_limit_fb>None</expected_upper_limit_fb>
            <lumi_fb-1>35.9</lumi_fb-1>
            <maxcond>0.0</maxcond>
            <r>1.5212733123965445</r>
            <r_expected>None</r_expected>
            <theory_prediction_fb>11.687259769407875</theory_prediction_fb>
            <upper_limit_fb>7.682550975009415</upper_limit_fb>
        </ExptRes>
```


SLHA Output

An SLHA-type output format is also available containing a summary of the *theory predictions* and *missing topologies*. The file contains the SLHA-type blocks: *SModelS_Settings*, *SModelS_Exclusion*, *SModelS_Missing_Topos*, *SModelS_Outside_Grid*, *SModelS_Long_Cascade* and *SModelS_Asymmetric_Branches*. Below we give a description of each block together with a sample output.

- information about the main input parameters:

```
BLOCK SModelS_Settings
 0 v1.2.0                    #SModelS version
 1 1.2.0                     #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 0.01                      #sigmacut [fb]
```

- 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 the model is excluded, all results with r-value greater than one are shown. 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 1.521E+00                      #r value
 1 2 1.521E+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
```

- a list of missing topologies (up to 10) and their weights (if **testCoverage** = True):

```
BLOCK SModelS_Missing_Topos #sqrts[TeV] weight[fb] description
 0 13  1.482E+02 [[[jet],[W]],[[jet,jet],[W]]] ('MET', 'MET')
 1 13  1.445E+02 [[[jet,jet],[W]],[[jet],[jet,jet],[W]]] ('MET', 'MET')
 2 13  1.047E+02 [[[b,t],[W]],[[jet,jet],[W]]] ('MET', 'MET')
```

- a list of topologies which are outside the *experimental results* grid (if **testCoverage** = True):

```
BLOCK SModelS_Outside_Grid #sqrts[TeV] weight[fb] description
 0 13  1.590E+00 [[[jet]],[[t,t]]] ('MET', 'MET')
 1 13  1.438E+00 [[[t],[W]],[[t],[W]]] ('MET', 'MET')
```

- a list of topologies with long cascade decays (if **testCoverage** = True):

```
BLOCK SModelS_Asymmetric_Branches #Mother1 Mother2 Weight[fb] allMothers
 0 1000021 1000021  5.297E+02 [[1000021,1000021]]
 1 1000002 1000021  4.149E+02 [[1000002,1000021],[1000004,1000021]]
```

- a list of topologies with asymmetric branch decays (if **testCoverage** = True):

```
 9 13  6.214E+01 [[[jet],[W]],[[jet,jet],[higgs]]] ('MET', 'MET')

BLOCK SModelS_Outside_Grid #sqrts[TeV] weight[fb] description
 0 13  1.590E+00 [[[jet]],[[t,t]]] ('MET', 'MET')
```

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`)
- 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 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`)

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

1.7 SModelS Code Documentation

These pages constitute the SModelS code documentation.

Contents

theory package

Submodules

theory.auxiliaryFunctions module

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.**distance**(*xmass1*, *xmass2*)
Define distance between two mass positions in upper limit space. The distance is defined as d = 2*|xmass1-xmass2|/(xmass1+xmass2).

Parameters

- **xmass1** – upper limit value (in fb) for the mass1
- **xmass2** – upper limit value (in fb) for the mass2

Returns relative mass distance in upper limit space

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.**massAvg**(*massList*, *method='weighted'*, *weights=None*)
Compute the average mass of massList according to method.

If method=weighted but weights were not properly defined, switch method to harmonic. If massList contains a zero mass, switch method to mean.

Parameters

- **method** – possible values: harmonic, mean, weighted
- **weights** – weights of elements (only for weighted average)

theory.auxiliaryFunctions.**massPosition**(*mass*, *txdata*)
Give mass position in upper limit space. Use the analysis experimental limit data. :param txdata: TxNameData object holding the data and interpolation

theory.branch module

class theory.branch.**Branch**(*info=None*, *finalState=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]').

Variables

- **masses** – list of masses for the intermediate states
- **particles** – list of particles (strings) for the final states
- **PIDs** – a list of the pdg numbers of the intermediate states appearing in the branch. If the branch represents more than one possible pdg list, PIDs will correspond to a nested list (PIDs = [[pid1,pid2,...],[pidA,pidB,...])
- **maxWeight** – weight of the branch (XSection object)

copy()

Generate an independent copy of self. Faster than deepcopy.

Returns Branch object

decayDaughter(*brDictionary*, *massDictionary*)

Generate a list of all new branches generated by the 1-step cascade decay of the current branch daughter.

Parameters

- **brDictionary** – dictionary with the decay information for all intermediate states (values are br objects, see pyslha)
- **massDictionary** – dictionary containing the masses for all intermediate states.

Returns list of extended branches (Branch objects). Empty list if daughter is stable or if daughterID was not defined.

getInfo()

Get branch topology info from particles.

Returns dictionary containing vertices and number of final states information

getLength()

Returns the branch length (number of R-odd particles).

Returns length of branch (number of R-odd particles)

particlesMatch(*other*)

Compare two Branches for matching particles, allow for inclusive particle labels (such as the ones defined in particles.py). Includes the final state in the comparison.

Parameters **other** – branch to be compared (Branch object)

Returns True if branches are equal (particles and masses match); False otherwise.

setFinalState(*finalState=None*)

If finalState = None, define the branch final state according to the PID of the last R-odd particle appearing in the cascade decay. Else set the final state to the finalState given :parameter finalState: String defining the final state

setInfo()

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

sortParticles()
Sort the particles inside each vertex

class theory.branch.**InclusiveBranch**
Bases: *theory.branch.Branch*

A branch wildcard class. It will return True when compared to any other branch object with the same final state.

getInfo()
Get branch topology info from particles.

Returns dictionary containing vertices and number of final states information

class theory.branch.**InclusiveInt**
Bases: int

A integer wildcard class. It will return True when compared to any other integer object.

class theory.branch.**InclusiveList**
Bases: list

A list wildcard class. It will return True when compared to any other list object.

theory.branch.**decayBranches**(*branchList, brDictionary, massDictionary, sigcut=0.00E+00 [fb]*)
Decay all branches from branchList until all unstable intermediate states have decayed.

Parameters

- **branchList** – list of Branch() objects containing the initial mothers
- **brDictionary** –

  dictionary with the decay information for all intermediate states (values are br objects, see pyslha).

  It may also contain information about long-lived particles.
- **massDictionary** – dictionary containing the masses for all intermediate states.
- **promptDictionary** – optional dictionary with the fraction of prompt and non-prompt decays. Allows to deal with quasi-stable or long-lived particles If not given, all particles are considered to always decay promptly or to be stable.
- **sigcut** – minimum sigma*BR to be generated, by default sigcut = 0. (all branches are kept)

Returns list of branches (Branch objects)

theory.clusterTools module

class theory.clusterTools.**ElementCluster**
Bases: object

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

Variables **elements** – list of elements in the cluster (Element objects)

getAvgMass()
Return the average mass of all elements belonging to the cluster. If the cluster does not refer to a TxName (i.e. in efficiency map results) AND the cluster contains more than one element (assuming they differ in the masses), returns None.

Returns average mass array

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

getIDs()
Return list of all element IDs appearing in the cluster :return: list of element IDs

getPIDs()
Return the list of all PIDs appearing in all elements in the cluster, i.e. [[[pdg1, pdg2,...],[pdg3,pdg4,...]], [[pdg1', pdg2',...],[pdg3',pdg4',...]]

Returns list of PIDs

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)

class theory.clusterTools.**IndexCluster**(*massMap=None*, *posMap=None*, *wMap=None*, *indices={}*, *txdata=None*)
Bases: object

An instance of this class represents a cluster storing element indices. This auxiliary class is used to store element indices and positions in upper limit space. It is only used by the clustering algorithm.

Variables

- **indices** – list of integers mapping the cluster elements to their position in the list (1st element -> index 0, 2nd element -> index 1,...)
- **avgPosition** – position in upper limit space for the cluster average mass
- **massMap** – dictionary with indices as keys and the corresponding element mass as values
- **positionMap** – dictionary with indices as keys and the corresponding element position in upper limit space as values
- **weightMap** – dictionary with indices as keys and the corresponding element weight as values
- **txdata** – TxNameData object to be used for computing distances in UL space

add(*iels*)
Add an index or a list of indices to the list of indices and update the avgPosition value.

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

remove(*iels*)
Remove an index or a list of indices to the list of indices and update the avgPosition value.

theory.clusterTools.**clusterElements**(*elements*, *maxDist*)
Cluster the original elements according to their mass distance.

Parameters

- **elements** – list of elements (Element objects)
- **txname** – TxName object to be used for computing distances in UL space
- **maxDist** – maximum mass distance for clustering two elements

Returns list of clusters (ElementCluster objects)

theory.clusterTools.**groupAll**(*elements*)
Create a single cluster containing all the elements. Skip mother elements which contain the daughter in the list (avoids double counting).

Parameters **elements** – List of Element objects

Returns ElementCluster object containing all unique elements

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, paritcle ids, center of mass, order and label).

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

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).

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.

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

combineWith(*newXsecs*)
Add a new list of cross sections.

If the new cross sections already appear (have same order and sqrts), add its value to the original value, otherwise append it to the list. The particle IDs are ignored when adding cross sections. Hence, they are set to (None, None) if any cross sections are combined.

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

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.element module

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

Variables

- **branches** – list of branches (Branch objects)
- **weight** – element weight (cross-section * BR)
- **motherElements** – only for elements generated from a parent element by mass compression, invisible compression,etc. Holds a pair of (whence, mother element), where whence describes what process generated the element

checkConsistency()
Check if the particles defined in the element exist and are consistent with the element info.

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

combineMotherElements(*el2*)
Combine mother elements from self and el2 into self

Parameters **el2** – element (Element Object)

combinePIDs(*el2*)
Combine the PIDs of both elements. If the PIDs already appear in self, do not add them to the list.

Parameters **el2** – element (Element Object)

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)

getDaughters()
Get a pair of daughter IDs (PDGs of the last intermediate state appearing the cascade decay), i.e. [[pdgLSP1,pdgLSP2]] Can be a list, if the element combines several daughters: [[pdgLSP1,pdgLSP2], [pdgLSP1',pdgLSP2']]

Returns list of PDG ids

getEinfo()
Get element topology info from branch topology info.

Returns dictionary containing vertices and number of final states information

getFinalStates()
Get the array of particles in the element.

Returns list of particle strings

getMasses()
Get the array of masses in the element.

Returns list of masses (mass array)

getMothers()
Get a pair of mother IDs (PDGs of the first intermediate state appearing the cascade decay), i.e. [[pdgMOM1,pdgMOM2]] Can be a list, if the element combines several mothers: [[pdgMOM1,pdgMOM2], [pdgMOM1',pdgMOM2']]

Returns list of PDG ids

getPIDs()
Get the list of IDs (PDGs of the intermediate states appearing the cascade decay), i.e. [[[pdg1,pdg2,...],[pdg3,pdg4,...]]]. The list might have more than one entry if the element combines different pdg lists: [[[pdg1,pdg2,...],[pdg3,pdg4,...]], [[pdg1',pdg2',...],[pdg3',pdg4',...]], ...]

Returns list of PDG ids

getParticles()
Get the array of particles in the element.

Returns list of particle strings

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

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;

particlesMatch(*other*, *branchOrder=False*)
Compare two Elements for matching particles only. Allow for inclusive particle labels (such as the ones defined in particles.py) and includes final state comparison. If branchOrder = False, check both branch orderings.

Parameters

- **other** – element to be compared (Element object)
- **branchOrder** – If False, check both orderings, otherwise check the same branch ordering

Returns True, if particles match; False, else;

setFinalState(*finalStates*)
If finalStates = None, define the element final states according to the PID of the last R-odd particle appearing in the cascade decay. Else set the final states according to the finalStates list (must match the branch ordering)

Parameters **finalStates** – List with final state labels (must match the branch ordering)

setMasses(*mass*, *sameOrder=True*, *opposOrder=False*)
Set the element masses to the input mass array.

Parameters

- **mass** – list of masses ([[masses for branch1],[masses for branch2]])
- **sameOrder** – if True, set the masses to the same branch ordering If True and opposOrder=True, set the masses to the smaller of the two orderings.
- **opposOrder** – if True, set the masses to the opposite branch ordering. If True and sameOrder=True, set the masses to the smaller of the two orderings.

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.lheDecomposer module

theory.lheDecomposer.**decompose**(*lhefile, inputXsecs=None, nevts=None, doCompress=False, doInvisible=False, minmassgap=-1.00E+00 [GeV]*)
Perform LHE-based decomposition.

Parameters

- **lhefile** – LHE file with e.g. pythia events, may be given as URL (though http and ftp only)
- **inputXsecs** – xSectionList object with cross sections for the mothers appearing in the LHE file. If None, use information from file.
- **nevts** – (maximum) number of events used in the decomposition. If None, all events from file are processed.
- **doCompress** – mass compression option (True/False)
- **doInvisible** – invisible compression option (True/False)

- **minmassgap** – minimum mass gap for mass compression (only used if doCompress=True)

Returns list of topologies (TopologyList object)

theory.lheDecomposer.**elementFromEvent**(*event*, *weight=None*)
Creates an element from a LHE event and the corresponding event weight.

Parameters

- **event** – LHE event
- **weight** – event weight. Must be a XSectionList object (usually with a single entry) or None if not specified.

Returns element

theory.lheReader module

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

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

close()
Close the lhe file, if open.

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.**Particle**
Bases: object

An instance of this class represents a particle.

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.particleNames module

class theory.particleNames.**InclusiveStr**
Bases: str

A string wildcard class. It will return True when compared to any other string.

theory.particleNames.**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.particleNames.**getFinalStateLabel**(*pid*)

Given the particle PID, returns the label corresponding to its final state (e.g. 1000022 -> MET, 1000023 -> HSCP,...) :parameter pid: PDG code for particle (must appear in particles.py) :return: Final state string (e.g. MET, HSCP,...)

theory.particleNames.**getName**(*pdg*)

Convert pdg number to particle name according to the dictionaries rOdd and rEven.

Returns particle name (e.g. gluino, mu-, ...)

theory.particleNames.**getPdg**(*name*)

Convert a name to the pdg number according to the dictionaries rOdd and rEven.

Returns particle pdg; None, if name could not be resolved

theory.particleNames.**simParticles**(*plist1*, *plist2*, *useDict=True*)

Compares two lists of particle names. Allows for dictionary labels (Ex: L = l, l+ = l, l = l-,...). Ignores particle ordering inside the list

Parameters

- **plist1** – first list of particle names, e.g. ['l','jet']
- **plist2** – second list of particle names
- **useDict** – use the translation dictionary, i.e. allow e to stand for e+ or e-, l+ to stand for e+ or mu+, etc

Returns True/False if the particles list match (ignoring order)

theory.particleNames.**vertInStr**(*instring*)

Parses instring (or a list of strings) and returns the list of particle vertices appearing in instring.

Parameters **instring** – string containing elements (e.g. "[[['e+']],[['e-']]]+[[['mu+']],[['mu-']]]")

Returns list of elements appearing in instring in string format

theory.slhaDecomposer module

theory.slhaDecomposer.**decompose**(*slhafile, sigcut=1.00E-01 [fb], doCompress=False, doInvisible=False, minmassgap=-1.00E+00 [GeV], useXSecs=None*)

Perform SLHA-based decomposition.

Parameters

- **slhafile** – the slha input file. May be an URL (though http, ftp only).
- **sigcut** – minimum sigma*BR to be generated, by default sigcut = 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)
- **useXSecs** – optionally a dictionary with cross sections for pair production, by default reading the cross sections from the SLHA file.

Returns list of topologies (TopologyList object)

theory.slhaDecomposer.**writeIgnoreMessage**(*keys*, *rEven*, *rOdd*)

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.

Variables

- **analysis** – holds the analysis (ULanalysis or EManalysis object) to which the prediction refers
- **xsection** – xsection of the theory prediction (relevant cross section to be compared with the experimental limits). For EM-type analyses, it corresponds to sigma*eff, for UL-type analyses, eff is considered to be 1. It is a XSection object.
- **conditions** – list of values for the analysis conditions (only for upper limit-type analysis, e.g. analysis=ULanalysis)
- **mass** – mass of the cluster to which the theory prediction refers (only for upper limit-type analysis, e.g. analysis=ULanalysis)

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()

Return the type of dataset

describe()

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)

class theory.theoryPrediction.**TheoryPredictionList**(*theoryPredictions=None*)
Bases: object

An instance of this class represents a collection of theory prediction objects.

Variables **_theoryPredictions** – list of TheoryPrediction objects

append(*theoryPred*)

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.

Variables

- **vertnumb** – list with number of vertices in each branch

- **verparts** – list with number of final states in each branch
- **elementList** – list of Element objects with this common topology

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.

Variables **topos** – list of topologies (Topology objects)

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 included 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*)

Bases: object

Database object. Holds a list of ExpResult objects.

Variables

- ***base*** – path to the database (string)
- **force_load** – force loading the text database ("txt"), or binary database ("pcl"), dont force anything if None
- **expResultList** – list of ExpResult objects

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

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

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

- **analysisID** – 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
- **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?

updateBinaryFile()
write a binar db file, but only if necessary.

class experiment.databaseObj.**ExpResultList**(*expResList*)
Bases: object

Holds a list of ExpResult objects for printout.

Variables **expResultList** – list of ExpResult objects

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.

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

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*)

Bases: object

Holds the information to a data set folder (TxName objects, dataInfo,...)

checkForRedundancy()
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. same as self.getTxName(txname).getEfficiencyFor(m)

getID()
Return the dataset ID

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(*mass=None*, *expected=False*, *txnames=None*, *compute=False*, *alpha=0.05*, *deltas_rel=0.2*)
Returns the upper limit for a given mass and txname. If the dataset hold an EM map result the upper limit is independent of the input txname or mass.

Parameters

- **txname** – TxName object or txname string (only for UL-type results)
- **mass** – 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=None*)

Returns a list for the possible values appearing in the DataSet for the required attribute.

Parameters **attribute** – name of a field in the database (string). If not defined it will return a dictionary with all fields and their respective values

Returns list of values

likelihood(*nsig*, *deltas_rel=0.2*, *marginalize=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. :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. :returns: likelihood to observe nobs events (float)

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*)

Bases: object

Object containing the information and data corresponding to an experimental result (experimental conference note or publication).

Variables

- **path** – path to the experimental result folder (i.e. ATLAS-CONF-2013-047)
- **globalInfo** – Info object holding the data in <path>/globalInfo.txt
- **datasets** – List of DataSet objects corresponding to the dataset folders in <path>

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=None*)
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). If not defined it will return a dictionary with all fields and their respective values

Returns list of values or value

hasCovarianceMatrix()

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".

Variables **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

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=201*, *python='3.6.8 (default, Jan 24 2020, 02:37:00) n[GCC 7.4.0]'*)
Bases: object

cTime()

current_version = 201
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*)
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

getEfficiencyFor(*mass*)
For upper limit results, checks if the input mass falls inside the upper limit grid. If it does, returns efficiency = 1, else returns efficiency = 0. For efficiency map results, checks if the mass falls inside the efficiency map grid. If it does, returns the corresponding efficiency value, else returns efficiency = 0.

Parameters **element** – Element object

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

getValueFor(*massarray*, *expected=False*)
Access txnameData and txnameDataExp to get value for massarray.

Parameters

- **massarray** – mass array values (with units), i.e. [[100*GeV,10*GeV],[100*GeV,10*GeV]]
- **expected** – query self.txnameDataExp

hasElementAs(*element*)
Verify if the conditions or constraint in Txname contains the element. Check both branch orderings.

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*, *datatag*, *Id*, *accept_errors_upto=0.05*)
Bases: object

Holds the data for the Txname object. It holds Upper limit values or efficiencies.

computeV(*values*)
Compute rotation matrix _V, and triangulation self.tri

Parameters **values** – Nested array with the data values

countNonZeros(*mp*)
count the nonzeros in a vector

evaluateString(*value*)
Evaluate string.

Parameters **value** – String expression.

flattenArray(*objList*)
Flatten any nested list to a 1D list.

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

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

formatInput(*value*, *shapeArray*)
Format value according to the shape in shapeArray. If shapeArray contains entries = *, the corresponding entries in value will be ignored.

Parameters

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

Returns formatted array [[200.,100.]]

getDataShape(*value*)
Stores the data format (mass shape) and store it for future use. If there are wildcards (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(*massarray*)
Interpolates the value and returns the UL or efficiency for the respective massarray

Parameters **massarray** – mass array values (with units), i.e. [[100*GeV,10*GeV],[100*GeV,10*GeV]]

interpolate(*point*, *fill_value=nan*)

loadData(*value*)
Uses the information in value to generate the data grid used for interpolation.

onlyZeroValues()
check if the map is zeroes only

removeUnits(*value*)
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])

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

removeWildCards(*value*)
Remove all entries = '*' from value.

Parameters **value** – usually a list containing floats and '*' (e.g. [[200.,'*'],'*'],0.4],..)

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.**Uncovered**(*topoList*, *sumL=True*, *sumJet=True*, *sqrts=None*)
Bases: object

Object collecting all information of non-tested/covered elements :ivar topoList: sms topology list :ivar sumL: if true, sum up electron and muon to lepton, for missing topos :ivar sumJet: if true, sum up jets, for missing topos :ivar sqrts: Center of mass energy. If defined it will only consider cross-sections for this value. Otherwise the highest sqrts value will be used.

addPrevMothers(*el*)

fill(*topoList*)
Check all elements, categorise those not tested / missing, classify long cascade decays and asymmetric branches Fills all corresponding objects :ivar topoList: sms topology list

getAllMothers(*topoList*)
Find all IDs of mother elements, only most compressed element can be missing topology :ivar topoList: sms topology list

getAsymmetricXsec(*sqrts=None*)

getLongCascadeXsec(*sqrts=None*)

getMissingX(*el*)
Calculate total missing cross section of element, by recursively checking if mothers are covered :ivar el: Element :returns: missing cross section in fb as number

getMissingXsec(*sqrts=None*)
Calculate total missing topology cross section at sqrts. If no sqrts is given use self.sqrts :ivar sqrts: sqrts

getOutOfGridXsec(*sqrts=None*)

getOutsideX(*el*)
Calculate total outside grid cross section of element, by recursively checking if mothers are covered :ivar el: Element :returns: missing cross section in fb as number

getTotalXsec(*sqrts=None*)
Calculate total cross-section from decomposition (excluding compressed elements) :ivar sqrts: sqrts

hasAsymmetricBranches(*el*)
Return True if Element branches are not equal :ivar el: Element

hasLongCascade(*el*)
Return True if element has more than 3 particles in the decay chain :ivar el: Element

inOutsideGridMothers(*el*)

inPrevMothers(*el*)

isMissingTopo(*el*)
A missing topology is not a mother element, not covered, and does not have mother which is covered :ivar el: Element

class tools.coverage.**UncoveredClass**(*motherPIDs*, *el*)
Bases: object

Object collecting all elements contributing to the same uncovered class, defined by the mother PIDs. :ivar motherPIDs: PID of initially produces particles, sorted and without charge information :ivar el: Element

add(*motherPIDs*, *el*)
Add Element to this UncoveredClass object if motherPIDs match and return True, else return False :ivar motherPIDs: PID of initially produces particles, sorted and without charge information :ivar el: Element

combine(*other*)

getWeight()
Calculate weight at sqrts :ivar sqrts: sqrts

isSubset(*other*)
True if motherPIDs of others are subset of the motherPIDs of this UncoveredClass

class tools.coverage.**UncoveredClassifier**
Bases: object

Object collecting elements with long cascade decays or asymmetric branches. Objects are grouped according to the initially produced particle PID pair.

addToClasses(*el*)
Add Element in corresponding UncoveredClass, defined by mother PIDs. If no corresponding class in self.classes, add new UncoveredClass :ivar el: Element

combine()

getMotherPIDs(*el*)

getSorted(*sqrts*)
Returns list of UncoveredClass objects in self.classes, sorted by weight :ivar sqrts: sqrts for weight lookup

remove(*cl*)
Remove element where mother pids match exactly

class tools.coverage.**UncoveredList**(*sumL*, *sumJet*, *sqrts*)
Bases: object

Object to find and collect UncoveredTopo objects, plus printout functionality :ivar sumL: if true sum electrons and muons to leptons :ivar sumJet: if true, sum up jets :ivar sqrts: sqrts, for printout

addToTopos(*el*)
adds an element to the list of missing topologies if the element contributes to a missing topology that is already in the list, add weight to topology :parameter el: element to be added

generalName(*instr*)
generalize by summing over charges e, mu are combined to l :parameter instr: element as string :returns: string of generalized element

class tools.coverage.**UncoveredTopo**(*topo*, *contributingElements=[]*)
Bases: object

Object to describe one missing topology result / one topology outside the mass grid :ivar topo: topology description :ivar weights: weights dictionary

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.

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=None*, *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). If not defined it will return a dictionary with all fields and their respective values
- **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.externalPythonTools module

class tools.externalPythonTools.**ExternalPythonTool**(*importname*, *optional=False*)
Bases: object

An instance of this class represents the installation of unum. As it is python-only, we need this only for installation, not for running (contrary to nllfast or pythia).

checkInstallation()
The check is basically done in the constructor

compile()

installDirectory()
Just returns the pythonPath variable

pathOfExecutable()
Just returns the pythonPath variable

tools.interactivePlots module

class tools.interactivePlots.**DataHolder**(*smodelsFolder*, *slhaFolder*, *parameterFile*)
Bases: object

A simple class to store the required data for producing the interactive plots

fillWith(*smodelsDict*, *slhaData*)
Fill the dictionary (data_dict) with the desired data from the smodels output dictionary (smodelsDict) and the pyslha.Doc object slhaData

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.

loadParameters()
Reads the parameters from the plotting parameter file.

makePlots(*outFolder*)
Uses the data in self.data_dict to produce the plots.

Parameters **outFolder** – Path to the output folder.

tools.interactivePlots.**main**(*args*)
Create the interactive plots using the input from argparse

Parameters **args** – argparser.Namespace object containing the options for makePlots

tools.interactivePlots.**makePlots**(*smodelsFolder*, *slhaFolder*, *outputFolder*, *parameters*, *npoints*, *verbosity*)
Main interface for the interactive-plots.

Parameters

- **smodelsFolder** – Path to the folder containing the SModelS python output
- **slhaFolder** – Path to the folder 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)

Returns True if the plot creation was successfull

tools.interactivePlotsHelpers module

tools.interactivePlotsHelpers.**create_index_html**(*path_to_plots*, *plot_data*, *plot_title*, *plot_list*, *plot_descriptions*)
Fills the index.html file with links to the interactive plots.

tools.interactivePlotsHelpers.**data_frame_excluded_nonexcluded**(*data_frame_all*)
Generate sub data frames for excluded and non-excluded points

tools.interactivePlotsHelpers.**fill_hover**(*data_frame_all*, *SModelS_hover_information*, *slha_hover_information*, *ctau_hover_information*, *BR_hover_information*)

Generates the text of the hover, according to users's requests.

tools.interactivePlotsHelpers.**get_BR**(*data_dict*, *slhaData*, *BR_hover_information*, *BR_get_top*)

Gets the requested branching ratios from the slha file, that will go into de hover.

tools.interactivePlotsHelpers.**get_asymmetric_branches**(*data_dict*, *smodelsOutput*)

Extracts the asymmetric branches info from the .py output. If requested, the data will be appended on each corresponding list

tools.interactivePlotsHelpers.**get_ctau**(*data_dict*, *slhaData*, *ctau_hover_information*)

Computes the requested ctaus, that will go into de hover.

tools.interactivePlotsHelpers.**get_entry**(*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.**get_expres**(*data_dict*, *smodelsOutput*)

Extracts the Expres info from the .py output. If requested, the data will be appended on each corresponding list

tools.interactivePlotsHelpers.**get_long_cascades**(*data_dict*, *smodelsOutput*)

Extracts the Long cascade info from the .py output. If requested, the data will be appended on each corresponding list

tools.interactivePlotsHelpers.**get_missed_topologies**(*data_dict*, *smodelsOuptut*)

Extracts the Missed topologies info from the .py output. If requested, the data will be appended on each corresponding list

tools.interactivePlotsHelpers.**get_outside_grid**(*data_dict*, *smodelsOutput*)

Extracts the outside grid info from the .py output. If requested, the data will be appended on each corresponding list.

tools.interactivePlotsHelpers.**get_slha_data**(*slhaFile*)

Uses pyslha to read the SLHA file. Return a pyslha.Doc objec, if successful.

tools.interactivePlotsHelpers.**get_slha_file**(*smodelsDict*)

Returns the file name of the SLHA file corresponding to the output in smodelsDict

tools.interactivePlotsHelpers.**get_slha_hover_info**(*data_dict*, *slhaData*, *slha_hover_information*)

Gets the requested slha info from eachh slha file, to fill the hover.

tools.interactivePlotsHelpers.**get_variable**(*data_dict*, *slhaData*, *slha_hover_information*, *variable*)

Gets the variable from the slha file.

tools.interactivePlotsHelpers.**get_xy_axis**(*variable_x*, *variable_y*)

Retrieves the names of the x and y axis variables.

tools.interactivePlotsHelpers.**import_python_output**(*smodelsFile*)

Imports the smodels output from each .py file.

tools.interactivePlotsHelpers.**make_continuous_plots_all**(*cont_plots*, *x_axis*, *y_axis*, *path_to_plots*, *data_frame_all*, *plot_data*, *plot_title*, *variable_x*, *variable_y*, *plot_descriptions*)

Generate plots with continuous z axis variables, using all data points

tools.interactivePlotsHelpers.**make_continuous_plots_excluded**(*cont_plots*, *x_axis*, *y_axis*, *path_to_plots*, *data_frame_excluded*, *plot_data*, *plot_title*, *variable_x*, *variable_y*, *plot_descriptions*)

Generate plots with continuous z axis variables, using excluded data points

tools.interactivePlotsHelpers.**make_continuous_plots_nonexcluded**(*cont_plots*, *x_axis*, *y_axis*, *path_to_plots*, *data_frame_nonexcluded*, *plot_data*, *plot_title*, *variable_x*, *variable_y*, *plot_descriptions*)

Generate plots with continuous z axis variables, using non-excluded data points

tools.interactivePlotsHelpers.**make_data_frame**(*data_dict*)

Transform the main dictionary in a data frame.

tools.interactivePlotsHelpers.**make_discrete_plots_all**(*disc_plots*, *x_axis*, *y_axis*, *path_to_plots*, *data_frame_all*, *plot_data*, *plot_title*, *variable_x*, *variable_y*, *plot_descriptions*)

Generate plots with discrete z axis variables, using all data points

tools.interactivePlotsHelpers.**make_discrete_plots_excluded**(*disc_plots*, *x_axis*, *y_axis*, *path_to_plots*, *data_frame_excluded*, *plot_data*, *plot_title*, *variable_x*, *variable_y*, *plot_descriptions*)

Generate plots with discrete z axis variables, using excluded data points

tools.interactivePlotsHelpers.**make_discrete_plots_nonexcluded**(*disc_plots*, *x_axis*, *y_axis*, *path_to_plots*, *data_frame_nonexcluded*, *plot_data*, *plot_title*, *variable_x*, *variable_y*, *plot_descriptions*)

Generate plots with discrete z axis variables, using non-excluded data points

tools.interactivePlotsHelpers.**output_status**(*smodelsDict*)

Check the smodels output status in the file, if it's -1, it will append 'none' to each list in the dictionary.

tools.interactivePlotsHelpers.**plot_description**()

Generate a description for each plot.

tools.interactivePlotsHelpers.**separate_cont_disc_plots**(*plot_list*, *data_dict*)

Generate sub lists of plots with discrete and conitnuous z axis variables.

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*, *sigmacut=None*)
Run checks on the input file.

Parameters

- **inputFile** – path to input file
- **sigmacut** – sigmacut in fb

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.

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.**Qnumbers**(*pid*)
Bases: object

An instance of this class represents quantum numbers.

Get quantum numbers (spin*2, electrical charge*3, color dimension) from qNumbers.

class tools.ioObjects.**ResultList**(*theoPredictionsList=[]*, *maxcond=1.0*)
Bases: object

Class that collects a list of theory predictions plus the corresponding upper limits.

addTheoPrediction(*theoPred*, *maxcond*)
Add a result to the theoryPredictions, unless it violates maxcond.

Parameters

- **theoPred** – a Theory Prediction object to be added to ResultList
- **maxcond** – maximum condition violation

getBestExpected()
Find EM result with the highest expected R vaue. :returns: Theory Prediction object

getR(*theoPred*, *expected=False*)
Calculate R value.

Parameters **theoPred** – Theory Prediction object

Returns R value = weight / upper limit

isEmpty()
Check if outputarray is empty.

sort()
Reverse sort theoryPredictions by R value.

class tools.ioObjects.**SlhaStatus**(*filename, maxDisplacement=0.01, sigmacut=3.00E-02 [fb], 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

degenerateChi()
Check if chi01 is lsp and chipm1 is NLSP. If so, check mass splitting. This function is not used, the limit is arbitrary.

deltaMass(*pid1*, *pid2*)
Calculate mass splitting between particles with pid1 and pid2.

Returns mass difference

emptyDecay(*pid*)
Check if any decay is listed 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

findLSP(*returnmass=None*)
Find lightest particle (not in rEven).

Returns pid, mass of the lsp, if returnmass == True

findLonglivedParticles(*findLonglived*)
Find meta-stable particles that decay to visible particles and stable charged particles.

Returns status flag, message

findMissingDecayBlocks(*findMissingBlocks*)
For all non-rEven particles listed in mass block, check if decay block is written

Returns status flag and message

findNLSP(*returnmass=None*)
Find second lightest particle (not in rEven).

Returns pid ,mass of the NLSP, if returnmass == True

getDecayWidth(*pid*)
Get the decay-width for particle with pid, if it exists.

getDecayWidths()
Get all decay-widths as a dictionary {pid: width}.

getLifetime(*pid*, *ctau=False*)
Compute lifetime from decay-width for a particle with pid.

Parameters

- **pid** – PID of particle
- **ctau** – set True to multiply lifetime by c

Returns lifetime

hasXsec(*checkXsec*)
Check if XSECTION table is present in the slha file.

Parameters **checkXsec** – set True to run the check

Returns status flag, message

massDiffLSPandNLSP()
Get the mass difference between the lsp and the nlsp.

read()
Get pyslha output object.

sumBR(*pid*)
Calculate the sum of all branching ratios for particle with pid.

Parameters **pid** – PID of particle

Returns sum of branching ratios as given in the decay table for pid

testLSP(*checkLSP*)
Check if LSP is charged.

Parameters **checkLSP** – set True to run the check

Returns status flag, message

visible(*pid*, *decay=None*)
Check if pid is detectable. If pid is not known, consider it as visible. If pid not SM particle and decay = True, check if particle or decay products are visible.

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*, *jobnr*)
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
- **jobnr** – number of process, in parallel mode. mostly for debugging.

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.**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.

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.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.

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.

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.pythonTools module

class tools.pythonTools.**PythonToolWrapper**(*importname*)

Bases: object

An instance of this class represents the installation of unum. As it is python-only, we need this only for installation, not for running (contrary to nllfast or pythia).

checkInstallation()

The check is basically done in the constructor

compile()

installDirectory()

Just returns the pythonPath variable

pathOfExecutable()

Just returns the pythonPath variable

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.**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

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=10000*)
Bases: object

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 total signal yield.

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=10000*, *cl=0.95*)
Bases: object

debug_mode = False

ulSigma(*model*, *marginalize=False*, *toys=None*, *expected=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.

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.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.

add(*instance*)
Adds a tool by passing an instance to this method.

checkInstallation(*make=False*, *printit=True*, *long=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()

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*)

getSqrtses(*args*)
extract the sqrtses from argument list

queryCrossSections(*filename*)

writeToFile(*args*)

class tools.xsecComputer.**XSecComputer**(*maxOrder*, *nevents*, *pythiaVersion*)
Bases: object

cross section computer class, what else?

addHigherOrders(*sqrts*, *slhafile*)
add higher order xsecs

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

compute(*sqrts*, *slhafile*, *lhefile=None*, *unlink=True*, *loFromSlha=None*, *pythiacard=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 /etc/pythia.card

Returns XSectionList object

computeForBunch(*sqrtses*, *inputFiles*, *unlink*, *lOfromSLHA*, *tofile*, *pythiacard=None*)
compute xsecs for a bunch of slha files

computeForOneFile(*sqrtses*, *inputFile*, *unlink*, *lOfromSLHA*, *tofile*, *pythiacard=None*)
compute the cross sections for one file. :param sqrtses: list of sqrt{s} tu run pythia, as a unum (e.g. 7*TeV)

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

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 */

using namespace std;

int main( int argc, char * argv[] )
```

(continues on next page)

(continued from previous page)

```
{
  /** initialise SModelS, load database, second argument (installdir) must point to␣
↪smodels
   *  installation top-level directory */
  SModelS smodels ( "./parameters.ini", "../" );
  /** run over one single file or directory */
  int ret = smodels.run ( "test.slha" );
}
```

A sample Makefile is also provided. The python header files have to be installed.

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