FTUV–14–3904
IFIC–14–40
ADP-14-17/T875

FlexibleSUSY — A spectrum generator generator for supersymmetric models

Peter Athron[a], Jae-hyeon Park[b], Dominik Stöckinger[c], Alexander Voigt[c]

[a] *ARC Centre of Excellence for Particle Physics at the Tera-scale, School of Chemistry and Physics, University of Adelaide, Adelaide SA 5005 Australia*
[b] *Departament de Física Teòrica and IFIC, Universitat de València-CSIC, 46100, Burjassot, Spain*
[c] *Institut für Kern- und Teilchenphysik, TU Dresden, Zellescher Weg 19, 01069 Dresden, Germany*

Abstract

We introduce FlexibleSUSY, a Mathematica and C++ package, which generates a fast, precise C++ spectrum generator for any SUSY model specified by the user. The generated code is designed with both speed and modularity in mind, making it easy to adapt and extend with new features. The model is specified by supplying the superpotential, gauge structure and particle content in a SARAH model file; specific boundary conditions e.g. at the GUT, weak or intermediate scales are defined in a separate FlexibleSUSY model file. From these model files, FlexibleSUSY generates C++ code for self-energies, tadpole corrections, renormalization group equations (RGEs) and electroweak symmetry breaking (EWSB) conditions and combines them with numerical routines for solving the RGEs and EWSB conditions simultaneously. The resulting spectrum generator is then able to solve for the spectrum of the model, including loop-corrected pole masses, consistent with user specified boundary conditions. The modular structure of the generated code allows for individual components to be replaced with an alternative if available. FlexibleSUSY has been carefully designed to grow as alternative solvers and calculators are added. Predefined models include the MSSM, NMSSM, E_6SSM, USSM, R-symmetric models and models with right-handed neutrinos.

Keywords: sparticle, supersymmetry, Higgs, renormalization group equations
PACS: 12.60.Jv
PACS: 14.80.Ly

1. Program Summary

Program title: FlexibleSUSY
Program obtainable from: `http://flexiblesusy.hepforge.org/`
Distribution format: tar.gz
Programming language: `C++, Wolfram/Mathematica, FORTRAN, Bourne shell`
Computer: Personal computer
Operating system: Tested on Linux 3.x, Mac OS X
External routines: SARAH 4.0.4, Boost library, Eigen, LAPACK
Typical running time: 0.06-0.2 seconds per parameter point.
Nature of problem: Determining the mass spectrum and mixings for any supersymmetric model. The generated code must find simultaneous solutions to constraints which are specified at two or more different renormalization scales, which are connected by renormalization group equations

forming a large set of coupled first-order differential equations.
Solution method: Nested iterative algorithm and numerical minimization of the Higgs potential.
Restrictions: The couplings must remain perturbative at all scales between the highest and lowest boundary condition. FlexibleSUSY assumes that all couplings of the model are real (i.e. $CP-$conserving). Due to the modular nature of the generated code adaption and extension to overcome restrictions in scope is quite straightforward.

2. Introduction

Supersymmetry (SUSY) provides the only non-trivial way to extend the space-time symmetries of the Poincaré group [1, 2], leading many to suspect that SUSY may be realized in nature in some form. In particular supersymmetric extensions of the standard model (SM) where SUSY is broken at the TeV scale have been proposed to solve the hierarchy problem [3, 4, 5, 6, 7], allow gauge coupling unification [8, 9, 10, 11, 12] and predict a dark matter candidate which can fit the observed relic density [13, 14]. Such models have also been used for baryogenesis or leptogensis to solve the matter-anti-matter asymmetry of the universe and have been considered as the low energy effective models originating from string theory.

Detailed phenomenological studies have been carried out for scenarios within the minimal supersymmetric standard model (MSSM). Such work has been greatly aided by public spectrum generators for the MSSM [16, 17, 18, 19, 20], allowing fast and reliable exploration of the sparticle spectrum, mixings and couplings, which can be obtained from particular choices of breaking mechanism inspired boundary conditions and specified parameters. Beyond the MSSM there are also two public spectrum generators [21, 22] for the next to minimal supersymmetric standard model (NMSSM) [23] (or for recent reviews see [24, 25]).

None of the fundamental motivations of supersymmetry require minimality, and specific alternatives to (or extensions of) the MSSM are, for example, motivated by the μ-problem of the MSSM [26]; explaining the family structure (see e.g. [27]) or for successful baryogenesis or leptogenesis (see e.g. [28]). However constructing specialized tools to study all relevant models would require an enormous amount of work. So general tools which can automate this process and produce fast and reliable programs can greatly enhance our ability to understand and test non-minimal realizations of supersymmetry.

Recent experimental developments have also increased the relevancy of such a tool. From the recent 7 TeV and 8 TeV runs at the Large Hadron Collider (LHC) there have been two important developments. Firstly low energy signatures expected from such models, such as the classic jets plus missing transverse energy signature, have not been observed, substantially raising the lower limit on sparticle masses (see e.g. [29, 30]). No other signature of beyond the standard model (BSM) physics has been observed, leaving the fundamental questions which motivated BSM physics unanswered. Secondly ATLAS and CMS discovered [31, 32] a light Higgs of 125 GeV, within the mass range that could be accommodated in the MSSM but requiring stops which are significantly heavier than both the direct collider limits and indirect limits that appears in constrained models from the significantly higher limits on first and second generation squarks.

These developments motivate the exploration of non-minimal SUSY models which ameliorate the naturalness problems, as can happen in the E_6SSM [33, 34, 35, 36], USSM [37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49] or from other gauge extensions[51, 52, 50], by raising the tree level Higgs mass. At the same time they can also motivate models that are developed with a fresh perspective, based on other considerations. In both cases exploration of such models can be aided if it is possible to quickly create a fast spectrum generator. Currently there is only one option for this, a SPheno-like FORTRAN code which can be generated from SARAH [53, 54, 55, 56, 57].

FlexibleSUSY provides a much needed alternative to this with a structure which has been freshly designed to accommodate as general range of models as possible and to be easily adaptable to changing goals and new ideas. FlexibleSUSY is a Mathematica and C++ package which uses SARAH to create a fast, modular C++ spectrum generator for a user specified SUSY model. The generated code structure is designed to be as flexible as possible to accommodate different types

of extensions and due to its modular nature it is easy to modify, add new features and combine with other programs. The generated code has been extensively tested against well known spectrum generators. As well as providing a solution for new SUSY models, the generated MSSM and NMSSM codes offer a modern and fast alternative to the existing public spectrum generators.

In Section 3 we describe the program in more detail and explain our design goals. In Section 4 information on how to download and compile the code may be found along with details on how to get started quickly. In Section 5 we describe how the user can create a new FlexibleSUSY model file. A detailed description of the structure and features of the generated code is then given in Section 6. In Section 7 we describe the various ways the code can be modified both at the meta code level by writing model files and at the C++ code level by modifying the code or adding new modules. Finally in Section 8 we describe detailed comparisons between our generated code and existing public spectrum generators as well as against the SPheno-like FORTRAN code which can be created using SARAH.

3. Overview of the program and design goals

To study the properties of SUSY models programs are needed which numerically calculate the pole masses and couplings of the SUSY particles given a set of theory input parameters. The output of these so-called spectrum generators can be transferred to programs which calculate further observables such as branching ratios or the dark matter relic density.

In order to create a spectrum generator the SUSY model must be defined by specifying the gauge group, the field content and mixings as well as the superpotential and the soft-breaking terms. From this information the renormalization group equations, mass matrices, self-energies, tadpole diagrams and electroweak symmetry breaking (EWSB) conditions have to be derived. These expressions must then be combined in a computer program to allow for a numeric calculation of the mass spectrum. In addition most SUSY models require boundary conditions for the model parameters at a low and a high scale. For example in the CMSSM mSUGRA boundary conditions for the soft-breaking parameters are imposed at the gauge coupling unification scale. Furthermore, at the Z mass scale the CMSSM is matched to the Standard Model, which implies conditions for the gauge and Yukawa couplings. The so defined boundary value problem must be solved numerically until a set of model parameters has been found consistent with all user-supplied boundary conditions.

FlexibleSUSY is a Mathematica and C++ package designed to create a fast and easily adaptable spectrum generator in C++ for any SUSY model. The user specifies the model by giving the superfield content, superpotential, gauge symmetries and mass mixings in form of SARAH model files. The boundary conditions on the model parameters must be specified in a separate `FlexibleSUSY.m.in` file. Based on this information FlexibleSUSY uses SARAH to obtain tree-level expressions for the mass matrices and electroweak symmetry breaking conditions, one-loop self energies, one-loop tadpoles corrections and two-loop renormalization group equations (RGEs) for the model. Additional corrections which have been calculated elsewhere, such as two-loop corrections to the Higgs masses[1] may be added by the user. These algebraic expressions are converted into C++ code and are put into classes with well-defined interfaces to allow for easy exchange, extension and reuse of the modules. All of these classes are finally combined to a complete spectrum generator, which solves the

[1] By default FlexibleSUSY has two-loop corrections to the Higgs masses for the MSSM [58, 59, 60, 61, 62] and NMSSM [63] in FORTRAN files supplied by Pietro Slavich. These are the same corrections which are implemented in many of the public spectrum generators.

user-defined boundary value problem. For this task FlexibleSUSY uses some parts of Softsusy [16], the very fast Eigen library [64], augmented by LAPACK, as well as the GNU scientific library and the Boost library to create numerical routines which solve the RGEs and boundary conditions simultaneously. If a solution has been found the pole mass spectrum is eventually calculated using full one-loop self-energies (and leading two-loop Higgs self-energy contributions for the MSSM and NMSSM).

Design goals

Since the calculation of the pole mass spectrum in a SUSY model is a non-trivial task, FlexibleSUSY is designed with the following points in mind:

Modularity. The large variety of supersymmetric models and potential investigations makes it likely that the user wants to modify the generated spectrum generator source code or reuse components in further programs. FlexibleSUSY offers two levels to influence the code: (i) On the Mathematica model file level the model itself or GUT/weak scale boundary conditions as well as input and output parameters can be controlled (see Section 4.3 and Section 7.1 for examples). (ii) In particular FlexibleSUSY uses C++ object orientation features to modularize the source code so that it is sharply divided into building blocks performing distinct duties. This modular architecture makes it easy for the user to modify, reuse, replace or extend the individual components (see Section 7.2 for examples). An important application of this concept are the boundary conditions, for which the C++ level offers a wider range of possibilities. The boundary conditions solver provides a plugin mechanism via a common `Constraint` interface, which allows a user to exchange or add boundary conditions at any scale. To realize this, all (derived) constraint objects are intentionally kept outside the solver. Despite being independent of one another, they can fit together with the aid of class inheritance. An elaborate example of a tower of effective field theories and multiple matching scales is presented in Section 7.2.1. Alternatively, the modular structure makes it straightforward to take FlexibleSUSY generated code for e.g. RGEs or self-energies and reuse it in an existing code for some other purpose. Conversely, it is also easy to include code from elsewhere into the spectrum generator. For an example see Section 7.2.2.

Speed. Exploring the parameter space of supersymmetric models with a high number of free parameters is quite time consuming. Therefore FlexibleSUSY aims to produce spectrum generators with a short run-time. The two most time consuming parts of a SUSY spectrum generator are usually the calculation of the two-loop β-functions and the pole masses of mixed particles:

- *Calculation of the β-functions:* The RG solving algorithms usually need $O(10)$ iterations between the high and the low scale to find a set of parameters consistent will all boundary conditions with a 0.01% precision goal. During each iteration the Runge-Kutta algorithm needs to calculate all β-functions $O(50)$ times. Most two-loop β-functions involve $O(50)$ matrix multiplications and additions. All together one arrives at $O(10^4)$ matrix operations. To optimize these, FlexibleSUSY uses the fast linear algebra package http://eigen.tuxfamily.org. Eigen uses C++ expression templates to remove temporary objects and enable lazy evaluation of the expressions. It supports explicit vectorization, and provides fixed-size matrices to avoid dynamic memory allocation. All of these features in combination result in very fast code for the calculation of the β-functions in FlexibleSUSY.

- *Calculation of the pole masses:* The second most time consuming part is the precise calculation of the pole masses for mixed particles. For each particle ψ_k in a multiplet the full self-energy matrix $\Sigma^{\psi}_{ij}(p = m^{\text{tree}}_{\psi_k})$ has to be evaluated. Each self-energy matrix entry again involves the calculation of $O(50)$ Feynman diagrams, each involving the calculation of vertices and a loop-function. All in all, one arrives at $O(500)$ Feynman diagrams and $O(10^4)$ loop function evaluations. To speed up the calculation of the pole masses FlexibleSUSY makes use of multi-threading, where each pole mass is calculated in a separate thread. This allows the operating system to distribute these calculations among different CPU cores. With this technique one can gain a speed-up of 20–30%.

Alternative boundary value problem solvers. Furthermore, the standard algorithm which solves the user-defined boundary value problem via a fixed-point iteration is not guaranteed to converge in all regions of the model parameter space. Therefore, FlexibleSUSY has been intentionally designed to allow for alternative solvers to search for solutions in such critical parameter regions. A subsequent release with an alternative solver is already planned.

Towers of effective theories. In FlexibleSUSY the standard fixed-point iteration solver has been generalized to handle towers of models (effective theories), which are matched at intermediate scales. An example of such a tower construction will be given in Section 7.2.1, where right-handed neutrinos are integrated out at the see-saw scale, between the SUSY and the GUT scale.

4. Quick start

4.1. Requirements

FlexibleSUSY can be downloaded from `http://flexiblesusy.hepforge.org`. To create a custom spectrum generator the following requirements are necessary:

- Mathematica, version 7 or higher
- SARAH, version 4.0.4 or higher `http://sarah.hepforge.org`
- C++11 compatible compiler (g++ 4.4.7 or higher, clang++ 3.1 or higher, icpc 12.1 or higher)
- FORTRAN compiler (gfortran, ifort etc.)
- Eigen library, version 3.1 or higher `http://eigen.tuxfamily.org`
- Boost library, version 1.36.0 or higher `http://www.boost.org`
- GNU scientific library `http://www.gnu.org/software/gsl`
- an implementation of LAPACK `http://www.netlib.org/lapack` such as ATLAS `http://math-atlas.sourceforge.net` or Intel Math Kernel Library `http://software.intel.com/intel-mkl`

Optional:

- Looptools, version 2.8 or higher `http://www.feynarts.de/looptools`

4.2. Downloading FlexibleSUSY and generating a first spectrum generator

FlexibleSUSY can be downloaded as a gzipped tar file from `http://flexiblesusy.hepforge.org`. To download and install version 1.0.0 run:

```
$ wget https://www.hepforge.org/archive/flexiblesusy/FlexibleSUSY-1.0.0.tar.gz
$ tar -xf FlexibleSUSY-1.0.0.tar.gz
$ cd FlexibleSUSY-1.0.0
```

A CMSSM spectrum generator can be created with the following three commands:

```
$ ./createmodel --name=MSSM
$ ./configure --with-models=MSSM
$ make
```

The first command creates the model directory `models/MSSM/` together with a CMSSM model file. The `configure` script checks the system requirements and creates the `Makefile`. See `./configure --help` for more options. Executing `make` will start Mathematica to generate the spectrum generator and compile it. The resulting executable can be run like this:

```
$ cd models/MSSM
$ ./run_MSSM.x --slha-input-file=LesHouches.in.MSSM
```

When executed, the spectrum generator tries to find a set of $\overline{\text{DR}}$ model parameters consistent with all CMSSM boundary conditions for the parameter point given in the SLHA input file `model_files/MSSM/LesHouches.in.MSSM`. Afterwards, the pole mass spectrum and mixing matrices are calculated and written to the standard output in SLHA format [65, 66]. For the parameter point given in the above example the calculated pole mass spectrum reads

```
Block MASS
   1000021     1.15236966E+03   # Glu
   1000024     3.85774334E+02   # Cha_1
   1000037     6.50460073E+02   # Cha_2
        25     1.14766149E+02   # hh_1
        35     7.06792640E+02   # hh_2
        37     7.11388516E+02   # Hpm_2
        36     7.06523105E+02   # Ah_2
   1000012     3.51856376E+02   # Sv_1
   1000014     3.53042556E+02   # Sv_2
   1000016     3.53046504E+02   # Sv_3
   1000022     2.03889780E+02   # Chi_1
   1000023     3.85760714E+02   # Chi_2
   1000025     6.36544884E+02   # Chi_3
   1000035     6.50133768E+02   # Chi_4
   1000001     9.66656018E+02   # Sd_1
   1000003     1.00983181E+03   # Sd_2
   1000005     1.01651873E+03   # Sd_3
   2000001     1.01653005E+03   # Sd_4
   2000003     1.06089534E+03   # Sd_5
   2000005     1.06090238E+03   # Sd_6
   1000011     2.22570305E+02   # Se_1
```

```
   1000013     2.29864536E+02   # Se_2
   1000015     2.29888846E+02   # Se_3
   2000011     3.61946671E+02   # Se_4
   2000013     3.61950866E+02   # Se_5
   2000015     3.63136031E+02   # Se_6
   1000002     8.09787818E+02   # Su_1
   1000004     1.01454197E+03   # Su_2
   1000006     1.01981109E+03   # Su_3
   2000002     1.02015269E+03   # Su_4
   2000004     1.05807759E+03   # Su_5
   2000006     1.05808168E+03   # Su_6
```

4.3. Spectrum generators for alternative models

FlexibleSUSY already comes with plenty of predefined models: the CMSSM (simply called `MSSM`), the NMSSM [23] in it's Z_3-symmetric form (called `NMSSM`), Z_3-violating NMSSM (`SMSSM`), the USSM (`UMSSM`) [37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49], the NUHM E_6SSM (`E6SSM`) [67], the right-handed neutrino extended MSSM (`MSSMRHN`), the NUHM-MSSM (`NUHMSSM`) and the R-symmetric MSSM (`MRSSM`) [68]. See the content of `model_files/` for all predefined model files. For all these models spectrum generators can be generated easily like for the CMSSM in Section 4.2. The spectrum generator for the Z_3-symmetric NMSSM for example can be generated like this:

```
$ ./createmodel --name=NMSSM
$ ./configure --with-models=NMSSM
$ make
```

One of the design goals is modularity and the possibility to easily construct custom spectrum generators. The details of the customization can be found in Sections 5–7. As a simple example consider the NMSSM. The NMSSM variant above unifies all soft-breaking trilinear scalar couplings at the GUT scale. In order to relax this constraint and use a separate value for A_λ at the GUT scale one can edit the model file `model_files/NMSSM/FlexibleSUSY.m.in` and change the lines

```
EXTPAR = { {61, LambdaInput} };

HighScaleInput = {
   ...
   {T[\[Lambda]], Azero LambdaInput},
   ...
};
```

into

```
EXTPAR = { {61, LambdaInput},
           {63, ALambdaInput} };

HighScaleInput = {
   ...
   {T[\[Lambda]], ALambdaInput LambdaInput},
   ...
};
```

The value of A_λ at the GUT scale can then be set in the SLHA input file in the EXTPAR block entry 63 via

```
Block EXTPAR
   61    0.1                  # LambdaInput
   63    -100                 # ALambdaInput
```

5. Setting up a FlexibleSUSY model

A general (non-constrained) softly broken SUSY model is defined by the gauge group, the field content and mixings as well as the superpotential and the soft-breaking Lagrangian. In order to create a spectrum generator for such a SUSY model with FlexibleSUSY, the aforementioned model properties have to be defined in a SARAH model file. The SARAH model file can be put into the `sarah/<model>/` directory. See the SARAH manual [69, 57] for a detailed explanation of how to write such a model file. Note that SARAH already is distributed with a lot of predefined models, which can be used with FlexibleSUSY immediately.

The model boundary conditions are defined in the FlexibleSUSY model file `FlexibleSUSY.m`, which has to be located in the model directory `models/<model>/`. To add this the user should create a `FlexibleSUSY.m.in` file in the directory `model_files/<model>/`. When the `./createmodel` script is executed, the `FlexibleSUSY.m` file is created from the `model_files/<model-file-name>/FlexibleSUSY.m.in` file, where the directory `<model-file-name>` is the specified by the `--model-file=<model-file-name>` option. If no such option is given the directory matching the `--name=<model>` option is used. In either case the `FlexibleSUSY.m` file which is created is then automatically placed in the directory `models/<model>/`. Note that many predefined example model files can already be found in `model_files/`.

In the following it is explained how the boundary conditions can be defined on the basis of the CMSSM. The application to other models is straightforward. The CMSSM model file reads:

```
FSModelName = "@CLASSNAME@";

MINPAR = {
   {1, m0},
   {2, m12},
   {3, TanBeta},
   {4, Sign[\[Mu]]},
   {5, Azero}
};

EWSBOutputParameters = { B[\[Mu]], \[Mu] };

HighScale = g1 == g2;

HighScaleFirstGuess = 2.0 10^16;

HighScaleMinimum = 1.0 10^10; (* optional *)

HighScaleMaximum = 1.0 10^18; (* optional *)

HighScaleInput = {
   {T[Ye], Azero*Ye},
```

```
   {T[Yd], Azero*Yd},
   {T[Yu], Azero*Yu},
   {mHd2, m0^2},
   {mHu2, m0^2},
   {mq2, UNITMATRIX[3] m0^2},
   {ml2, UNITMATRIX[3] m0^2},
   {md2, UNITMATRIX[3] m0^2},
   {mu2, UNITMATRIX[3] m0^2},
   {me2, UNITMATRIX[3] m0^2},
   {MassB, m12},
   {MassWB, m12},
   {MassG, m12}
};

SUSYScale = Sqrt[M[Su[1]]*M[Su[6]]];

SUSYScaleFirstGuess = Sqrt[m0^2 + 4 m12^2];

SUSYScaleInput = {};

LowScale = SM[MZ];

LowScaleFirstGuess = SM[MZ];

LowScaleInput = {
   {Yu, Automatic},
   {Yd, Automatic},
   {Ye, Automatic},
   {vd, 2 MZDRbar / Sqrt[GUTNormalization[g1]^2 g1^2 + g2^2]
            Cos[ArcTan[TanBeta]]},
   {vu, 2 MZDRbar / Sqrt[GUTNormalization[g1]^2 g1^2 + g2^2]
            Sin[ArcTan[TanBeta]]}
};

InitialGuessAtLowScale = {
   {vd, SM[vev] Cos[ArcTan[TanBeta]]},
   {vu, SM[vev] Sin[ArcTan[TanBeta]]},
   {Yu, Automatic},
   {Yd, Automatic},
   {Ye, Automatic}
};

InitialGuessAtHighScale = {
   {\[Mu]   , 1.0},
   {B[\[Mu]], 0.0}
};

UseHiggs2LoopMSSM = True;
EffectiveMu = \[Mu];

OnlyLowEnergyFlexibleSUSY = False; (* default *)

PotentialLSPParticles = { Chi, Cha, Glu, Sv, Su, Sd, Se };

DefaultPoleMassPrecision = MediumPrecision;
HighPoleMassPrecision    = {hh, Ah, Hpm};
```

```
MediumPoleMassPrecision  = {};
LowPoleMassPrecision     = {};
```

The first line `FSModelName = "@CLASSNAME@";` will be replaced with `FSModelName = "<model>";` in the generated `FlexibleSUSY.m` file, where `<model>` is specified by the `--name=<model>` option for the `./createmodel` script. So the variable `FSModelName` then contains the name of the FlexibleSUSY model.

All non-Standard Model input variables must be specified in the lists `MINPAR` and `EXTPAR`. These two variables refer to the MINPAR and EXTPAR blocks in a SLHA input file [65]. The list elements are two-component lists where the first entry is the SLHA index in the MINPAR or EXTPAR block, respectively, and the second entry is the name of the input parameter. In the above example the input parameters are the universal soft-breaking parameters m_0, $M_{1/2}$, A_0 as well as $\tan\beta$ and $\operatorname{sign}\mu$.

Using the variable `EWSBOutputParameters` the user can specify the model parameters that are output of the electroweak symmetry breaking consistency conditions. When imposing the EWSB, FlexibleSUSY will adjust these parameters until the EWSB conditions are fulfilled. In the CMSSM example above these are the superpotential parameter μ and its corresponding soft-breaking parameter $B\mu$. In the NMSSM the parameters κ, $|v_s|$ and m_s^2 are usually chosen for this purpose.

Furthermore, the user has to specify three model constraints: low-scale, SUSY-scale and high-scale. In FlexibleSUSY they are named as `LowScale`, `SUSYScale` and `HighScale`. For each constraint there is (i) a scale definition (named after the constraint), (ii) an initial guess for the scale (concatenation of the constraint name and `FirstGuess`) and (iii) a list of parameter settings to be applied at the scale (concatenation of the constraint name and `Input`). Optionally a minimum and a maximum value for the scale can be given (concatenation of the constraint name and `Minimum` or `Maximum`, respectively). The latter avoids underflows or overflows of the scale value during the iteration. This is especially useful in models where the iteration is very unstable and the value of the scale is very sensitive to the model parameters. The meaning of the three constraints is the following:

- *High-scale constraint:* The high-scale constraint is usually the GUT-scale constraint, imposed at the scale where the gauge couplings g_1 and g_2 unify. The high-scale can be defined by an equation of the form `g1 == g2` or by a fixed numerical value. Note that FlexibleSUSY GUT-normalizes all gauge couplings. Thus, the high-scale definition takes the simple form `g1 == g2`. As a consequence in the calculation of the VEVs v_u and v_d from M_Z and $\tan\beta$ at the low-scale the GUT-normalization has to be taken into account, see the example above.

- *SUSY-scale constraint:* The SUSY-scale is the typical mass scale of the SUSY particle spectrum. At this scale FlexibleSUSY imposes the EWSB conditions and calculates the pole mass spectrum. The SUSY-scale, M_S, is usually defined as $M_S = \sqrt{m_{\tilde{t}_1} m_{\tilde{t}_2}}$. However, in the example above, where sfermion flavour violation is enabled, it has the value $M_S = \sqrt{m_{\tilde{u}_1} m_{\tilde{u}_6}}$, where $m_{\tilde{u}_1}$ and $m_{\tilde{u}_6}$ are the $\overline{\text{DR}}$ masses of the lightest and heaviest up-type squark, respectively.

- *Low-scale constraint:* The low-scale constraint is the constraint where the SUSY model is matched to the Standard Model. This is done by automatically calculating the gauge couplings g_i ($i = 1, 2, 3$) of the SUSY model from the known Standard Model quantities $\alpha_{\text{e.m.}}(M_Z)$, $\alpha_s(M_Z)$, M_Z, M_W. The details of this calculation are explained in Section 6.2.1. Currently this scale is fixed to be the Z pole mass scale M_Z. Optionally the Yukawa couplings y_f ($f = u, d, e$) can be calculated automatically from the known Standard Model fermion masses m_f by setting their values to `Automatic`. This automatic calculation is explained in Section 6.2.2.

The variables `LowScaleInput`, `SUSYScaleInput HighScaleInput`, which list the parameter settings for imposing the constraints can contain as elements any of the following:

- Two-component lists of the form `{parameter, value}`, which indicates that the `parameter` is set to `value` at the defined scale. If the `value` should be read from the SLHA input file, it must be written as `LHInput[value]`. Example:

```
SUSYScaleInput = {
   {mHd2, m0^2},
   {mHu2, LHInput[mHu2]}
};
```

  In this example the parameter `mHd2` is set to the value of `m0^2`, and `mHu2` is set to the value given in the SLHA input file in block `MSOFTIN`, entry 22 at the SUSY scale. The SLHA block names and keys for the MSSM and NMSSM are defined in SARAH's `parameters.m` file, see the SARAH manual or [55]. For the Standard Model Yukawa couplings `Yu`, `Yd`, `Ye` the value `Automatic` is allowed, which triggers their automatic determination from the known Standard Model quark and lepton masses, see Section 6.2.2.

- The function `FSMinimize[parameters, function]` can be given, where `parameters` is a list of model parameters and `function` is a function of these parameters. `FSMinimize[parameters, function]` will numerically vary the `parameters` until the `function` is minimized. Example:

```
FSMinimize[{vd,vu},
           (SM[MZ] - Pole[M[VZ]])^2 / STANDARDDEVIATION[MZ]^2 +
           (SM[MH] - Pole[M[hh[1]]])^2 / STANDARDDEVIATION[MH]^2]
```

  Here, the parameters `vu` and `vd` are varied until the function

$$\chi^2(v_d, v_u) = \frac{(\texttt{SM[MZ]} - m_Z^{\text{pole}})^2}{\sigma_{m_Z}^2} + \frac{(\texttt{SM[MH]} - m_{h_1}^{\text{pole}})^2}{\sigma_{m_h}^2} \tag{1}$$

  is minimal. The constants `SM[MZ]`, `SM[MH]`, σ_{m_Z} and σ_{m_h} are defined in `src/ew_input.hpp` to be

$$\texttt{SM[MZ]} = 91.1876, \qquad \texttt{SM[MH]} = 125.9, \tag{2}$$

$$\sigma_{m_Z} = 0.0021, \qquad \sigma_{m_h} = 0.4. \tag{3}$$

- The function `FSFindRoot[parameters, functions]` can be given, where `parameters` is a list of model parameters and `functions` is a list of functions of these parameters. `FSFindRoot[parameters, \ functions]` will numerical vary the `parameters` until the `functions` are zero. Example:

```
FSFindRoot[{vd,vu},
           {SM[MZ] - Pole[M[VZ]], SM[MH] - Pole[M[hh[1]]]}]
```

  Here, the parameters `vu` and `vd` are varied until the vector-valued function

$$f(v_d, v_u) = \begin{pmatrix} \texttt{SM[MZ]} - m_Z^{\text{pole}} \\ \texttt{SM[MH]} - m_{h_1}^{\text{pole}} \end{pmatrix} \tag{4}$$

is zero.

Finally, the user can set an initial guess for the model parameters at the low- and high-scale using the variables `InitialGuessAtLowScale` and `InitialGuessAtHighScale`, respectively. The gauge couplings will be guessed automatically at the low-scale from the known Standard Model parameters.

FlexibleSUSY allows to add leading two-loop contributions to the CP-even Higgs tadpoles and self-energies. For MSSM-like models (with two CP-even Higgs bosons, one CP-odd Higgs boson, one neutral Goldstone boson) these corrections can be enabled by setting `UseHiggs2LoopMSSM = \ True` in the model file and by defining the effective μ-term `EffectiveMu = \[Mu]`. This will add the zero-momentum corrections of the order $O(y_t^4 + y_b^2 y_t^2 + y_b^4)$, $O(y_t^2 g_3^2)$, $O(y_b^2 g_3^2)$, $O(y_\tau^4)$ from [58, 59, 60, 61, 62]. For NMSSM-like models (with three CP-even Higgs bosons, two CP-odd Higgs bosons, one neutral Goldstone boson) the two-loop contributions are enabled by setting `UseHiggs2LoopNMSSM = True` and by defining the effective μ-term like `EffectiveMu = \[Lambda] vS / \ Sqrt[2]`, for example. This will add the zero-momentum corrections of the order $O(y_t^2 g_3^2)$, $O(y_b^2 g_3^2)$ from [63], plus MSSM-like contributions of the order $O(y_\tau^4)$, $O(y_t^4 + y_t^2 y_b^2 + y_b^4)$ [59, 62].

One can create a pure low-energy model by setting `OnlyLowEnergyFlexibleSUSY = True`. In this case the high-scale constraint is ignored and only the low-scale and SUSY-scale constraints are used. All model parameters which are not specified in `MINPAR` or `EXTPAR` will then be read from the corresponding input blocks in the SLHA input file and will be set at the SUSY-scale. An example of such a pure low-energy model is the MRSSM, where the three gauge couplings do not unify at a common scale.

FlexibleSUSY can create the helper function `get_lsp()`, which finds the lightest supersymmetric particle (LSP). To have this function be created the model file variable `PotentialLSPParticles` must be set to a list of SUSY particles which are potential LSPs. In the model file example above, the particles `Chi`, `Cha`, `Glu`, `Sv`, `Su`, `Sd`, `Se` (neutralino, chargino, gluino, sneutrino, up-type squark, down-type squark, selectron) are considered to be LSP candidates.

6. Structure of the spectrum generator

In this section we explain the internals of FlexibleSUSY's automatically generated spectrum generator.

As mentioned in Section 3, FlexibleSUSY uses SARAH-generated expressions for the β-functions, mass matrices, self-energies and EWSB conditions plus the user-defined parameter boundary conditions to create a spectrum generator in C++. This program takes the Standard Model and user-defined input parameters and numerically solves the boundary value problem, which is defined by the RG equations and the boundary conditions. If a solution is found the pole mass spectrum is calculated.

In the following it is explained how this procedure is realized in FlexibleSUSY. As mentioned in Section 3 one of FlexibleSUSY's design goals is to create modular C++ code to allow for an easy exchange, extension and reuse of the generated modules. For this reason Section 6.1 first of all briefly describes the so-called C++ "model class" hierarchy, which contains the general model information, such as parameters, β-functions, $\overline{\text{DR}}$ mass spectrum, EWSB, self-energies, and the pole mass spectrum. Section 6.2 describes how boundary conditions on the model parameters are implemented in general at the C++ level. Subsections 6.2.1–6.2.3 then show the two concrete boundary conditions, which are always imposed: The matching of the model parameters to the Standard Model and the electroweak symmetry breaking. In Section 6.3 we describe the conventions

used to calculate the $\overline{\text{DR}}$ mass spectrum given a set of $\overline{\text{DR}}$ model parameters. Afterwards, in Section 6.4 the algorithm, which solves the user-defined boundary value problem is described on the basis of the CMSSM example given in Section 5. Finally, Section 6.5 explains how the pole mass spectrum is obtained from the $\overline{\text{DR}}$ model parameters after a solution to the boundary value problem has been found.

6.1. Model parameters and RGEs

The parameters of the model together with their RGEs, mass matrices, self-energies and EWSB equations are stored at the C++ level in the model class hierarchy, which is shown in the UML diagram in Figure 1.

The top of the hierarchy is formed by the `Beta_function` interface class, which defines the basic RGE running interface. It provides the interface function `run_to()`, which integrates the RGEs up to a given scale using an adaptive Runge-Kutta algorithm. This algorithm uses the pure virtual functions `get()`, `set()` and `beta()`, which need to be implemented by a derived class. The `get()` and `set()` functions return and set the model parameters in form of a vector, respectively. The `beta()` method returns the β-function for each parameter in form of a vector as well.

All model parameters and their β-functions are contained in the first and second derived classes. The structure of the β-functions of a general supersymmetric model [70, 71, 72, 73, 74, 75, 76, 77, 78, 79] allows to split these parameters into two classes:

1. *SUSY parameters:* gauge couplings, superpotential parameters and VEVs and
2. *soft-breaking parameters* [15]: soft linear scalar terms, soft bilinear scalar interactions, soft trilinear scalar interactions, soft gaugino mass terms and soft scalar squared masses.

The β-functions of the SUSY parameters in general depend only on the SUSY parameters and are independent of the soft-breaking parameters. However, the β-functions of the soft-breaking parameters depend on all model parameters in general. This property is reflected in the C++ code: The class `<model>_susy_parameters` directly inherits from `Beta_functions` and implements the β-functions of the SUSY parameters. The class of soft-breaking parameters `<model>_soft_parameters` in turn inherits from `<model>_susy_parameters` and implements the β-functions of the soft-breaking parameters in terms of all model parameters. The so constructed class hierarchy allows to (i) use the RGE running of all model parameters via the common `Beta_function` interface and to (ii) run the SUSY parameters independently of the soft-breaking parameters.

FlexibleSUSY creates these two classes from the model parameters defined in the SARAH model file. The corresponding one- and two-loop β-functions are calculated algebraically using SARAH's `CalcRGEs[]` routine, converted to C++ form and written into the corresponding `beta()` functions. These two classes then allow to use renormalization group running of all model parameters.

At the bottom of the hierarchy stands the actual model class, which uses the $\overline{\text{DR}}$ parameters from the parent classes to calculate $\overline{\text{DR}}$ and pole mass spectra. These two calculations are performed in the `calculate_DRbar_parameters()` and `calculate_pole_masses()` functions, which make use of the mass matrices and self-energies obtained from SARAH. The calculation of the pole mass spectrum will be explained in detail in Section 6.5. The resulting masses can be obtained by calling `get_physical()`. The `calculate_spectrum()` function combines these two spectrum computations into one call. In addition, the model class provides a `solve_ewsb()` method, which solves the electroweak symmetry breaking equations numerically at the loop level. This function is explained in the next section.

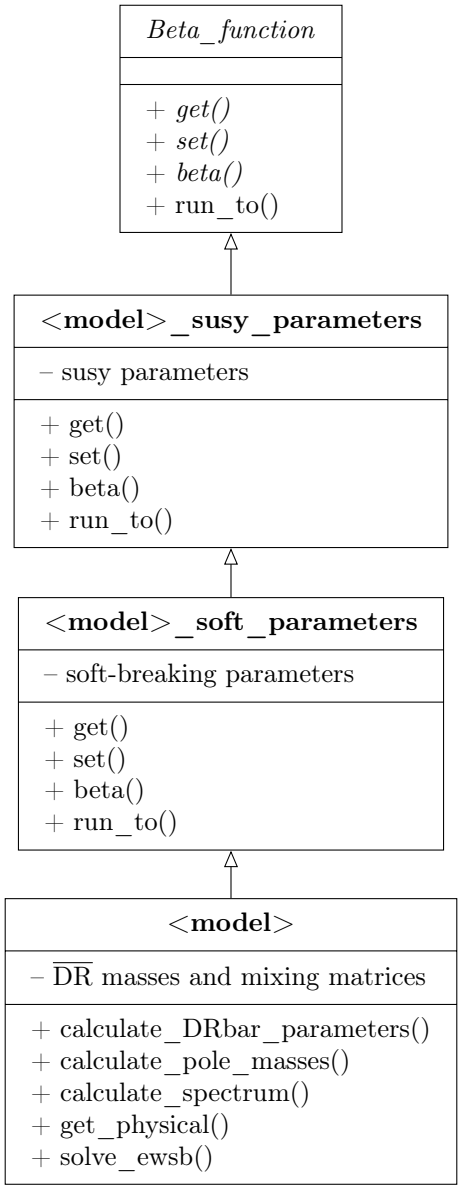

Figure 1: Model class hierarchy.

6.2. Boundary conditions

As described in Section 5, the user defines three boundary conditions in the FlexibleSUSY model file at the Mathematica level. These boundary conditions are converted to C++ form and are put into classes, which implement the common `Constraint<Two_scale>` interface. This interface has the form:

```
template<>
class Constraint<Two_scale> {
public:
   virtual ~Constraint() {}
   virtual void apply() = 0;
   virtual double get_scale() const = 0;
};
```

The `get_scale()` function is supposed to return the renormalization scale at which the constraint is to be imposed. The `apply()` method imposes the constraint by setting model parameters to values as chosen by the user. The three boundary condition classes are generated as follows:

- The *high-scale constraint* is intended to set boundary conditions on the model parameters at some very high scale, e.g. the GUT scale M_X. The high-scale is defined by the value given in the variable `HighScale`. In the CMSSM example model file in Section 5 it is defined to be the unification scale M_X where $g_1(M_X) = g_2(M_X)$. The `apply()` function is implemented by setting model parameters to the values defined in the `HighScaleInput` variable.

- The *SUSY-scale constraint* is intended to set boundary conditions at the mass scale M_S of the SUSY particles. The value of M_S is defined by the model file variable `SUSYScale`. In the example model file in Section 5 it is defined to be $\sqrt{m_{\tilde{u}_1} m_{\tilde{u}_6}}$, where $m_{\tilde{u}_1}$ is the mass of the lightest, and $m_{\tilde{u}_6}$ is the mass of the heaviest up-type squark. The `apply()` function for this constraint sets the model parameters to the values defined in `SUSYScaleInput`. Afterwards, `apply()` solves the EWSB equations at the loop level by adjusting the parameters given in `EWSBOutputParameters` such that the effective Higgs potential is minimized. See Section 6.2.3 for a more detailed description of the EWSB in FlexibleSUSY.

- The *low-scale constraint* is intended to match the SUSY model to the Standard Model at the scale M_Z. It does so by calculating the gauge couplings of the SUSY model from the known Standard Model quantities $\alpha^{(5),\overline{\text{MS}}}_{\text{e.m.,SM}}(M_Z)$, $\alpha^{(5),\overline{\text{MS}}}_{\text{s,SM}}(M_Z)$, M_Z and M_W. This calculation is explained in Section 6.2.1. Optionally, the Yukawa couplings of the SUSY model can be calculated automatically from the Standard Model fermion masses. See Section 6.2.2 for more details. In addition to the gauge and Yukawa couplings, the model parameter constraints given in `LowScaleInput` are imposed here.

6.2.1. Calculation of the gauge couplings $g_i(M_Z)$

The low-scale constraint automatically calculates the $\overline{\text{DR}}$ gauge couplings $g^{\overline{\text{DR}}}_{i,\text{susy}}(M_Z)$ in the SUSY model at the scale M_Z. It starts from the known electromagnetic and strong $\overline{\text{MS}}$ couplings in the Standard Model including only 5 quark flavours $\alpha^{(5),\overline{\text{MS}}}_{\text{e.m.,SM}}(M_Z) = 1/127.944$ and $\alpha^{(5),\overline{\text{MS}}}_{\text{s,SM}}(M_Z) = 0.1185$ [80]. These are converted to the electromagnetic and strong $\overline{\text{DR}}$ couplings in the SUSY

model $e^{\overline{\rm DR}}_{\rm susy}(M_Z)$ and $g^{\overline{\rm DR}}_{3,\rm susy}(M_Z)$ as

$$\alpha^{\overline{\rm DR}}_{\rm e.m.,susy}(M_Z) = \frac{\alpha^{(5),\overline{\rm MS}}_{\rm e.m.,SM}(M_Z)}{1-\Delta\alpha_{\rm e.m.,SM}(M_Z)-\Delta\alpha_{\rm e.m.,susy}(M_Z)}, \tag{5}$$

$$e^{\overline{\rm DR}}_{\rm susy}(M_Z) = \sqrt{4\pi\alpha^{\overline{\rm DR}}_{\rm e.m.,susy}(M_Z)}, \tag{6}$$

$$\alpha^{\overline{\rm DR}}_{\rm s,susy}(M_Z) = \frac{\alpha^{(5),\overline{\rm MS}}_{\rm s,SM}(M_Z)}{1-\Delta\alpha_{\rm s,SM}(M_Z)-\Delta\alpha_{\rm s,susy}(M_Z)}, \tag{7}$$

$$g^{\overline{\rm DR}}_{3,\rm susy}(M_Z) = \sqrt{4\pi\alpha^{\overline{\rm DR}}_{\rm s,susy}(M_Z)}. \tag{8}$$

The $\Delta\alpha_i(\mu)$ are threshold corrections and read

$$\Delta\alpha_{\rm e.m.,SM}(\mu) = \frac{\alpha_{\rm e.m.}}{2\pi}\left[\frac{1}{3}-\frac{16}{9}\log\frac{m_t}{\mu}\right], \tag{9}$$

$$\Delta\alpha_{\rm e.m.,susy}(\mu) = \frac{\alpha_{\rm e.m.}}{2\pi}\left[-\sum_{\text{susy particle } i} F_i T_i \log\frac{m_i}{\mu}\right], \tag{10}$$

$$\Delta\alpha_{\rm s,SM}(\mu) = \frac{\alpha_{\rm s}}{2\pi}\left[-\frac{2}{3}\log\frac{m_t}{\mu}\right], \tag{11}$$

$$\Delta\alpha_{\rm s,susy}(\mu) = \frac{\alpha_{\rm s}}{2\pi}\left[\frac{1}{2}-\sum_{\text{susy particle } i} F_i T_i \log\frac{m_i}{\mu}\right], \tag{12}$$

where the sums on the right-hand sides run over all electrically and color charged fields absent from the Standard Model. The constants T_i are the Dynkin indices of the representation of particle i with respect to the gauge group, and F_i are particle-type specific constants [81]

$$F_i = \begin{cases} 2/3 & \text{if particle } i \text{ is a Majorana fermion,} \\ 4/3 & \text{if particle } i \text{ is a Dirac fermion,} \\ 1/6 & \text{if particle } i \text{ is a real scalar,} \\ 1/3 & \text{if particle } i \text{ is a complex scalar.} \end{cases} \tag{13}$$

Afterwards, SARAH's expression for the Weinberg angle θ_W in terms of $M^{\overline{\rm DR}}_{W,\rm susy}(M_Z)$, $M^{\overline{\rm DR}}_{Z,\rm susy}(M_Z)$ is used to calculate θ_W in the SUSY model in the $\overline{\rm DR}$ scheme. In the MSSM, for example, it yields

$$\theta^{\overline{\rm DR}}_{W,\rm susy}(M_Z) = \arcsin\sqrt{1-\left(\frac{M^{\overline{\rm DR}}_{W,\rm susy}(M_Z)}{M^{\overline{\rm DR}}_{Z,\rm susy}(M_Z)}\right)^2}. \tag{14}$$

In a model with a Higgs triplet the relation looks like

$$\theta^{\overline{\rm DR}}_{W,\rm susy}(M_Z) = \arcsin\sqrt{1-\frac{\left(M^{\overline{\rm DR}}_{W,\rm susy}(M_Z)\right)^2 - g_2^2 v_T^2}{\left(M^{\overline{\rm DR}}_{Z,\rm susy}(M_Z)\right)^2}}, \tag{15}$$

where v_T is the vacuum expectation value of the scalar Higgs triplet field. The running $\overline{\rm DR}$ W and Z boson masses are calculated in each iteration from the corresponding pole masses as

$$\left(M_{W,\rm susy}^{\overline{\rm DR}}(M_Z)\right)^2 = M_W^2 + \mathrm{Re}\,\Pi_{WW}^T(p^2 = M_W^2, \mu = M_Z), \tag{16}$$

$$\left(M_{Z,\rm susy}^{\overline{\rm DR}}(M_Z)\right)^2 = M_Z^2 + \mathrm{Re}\,\Pi_{ZZ}^T(p^2 = M_Z^2, \mu = M_Z), \tag{17}$$

where $M_W = 80.404\,\mathrm{GeV}$ and $M_Z = 91.1876\,\mathrm{GeV}$ [80]. Having $e_{\rm susy}^{\overline{\rm DR}}(M_Z)$ and $\theta_{W,\rm susy}^{\overline{\rm DR}}(M_Z)$ allows to calculate the (GUT-normalized) $U(1)_Y$ and $SU(2)_L$ gauge couplings in the SUSY model. In the MSSM they read for instance

$$g_{1,\rm susy}^{\overline{\rm DR}}(M_Z) = \sqrt{\frac{5}{3}}\frac{e_{\rm susy}^{\overline{\rm DR}}(M_Z)}{\cos\theta_{W,\rm susy}^{\overline{\rm DR}}(M_Z)}, \tag{18}$$

$$g_{2,\rm susy}^{\overline{\rm DR}}(M_Z) = \frac{e_{\rm susy}^{\overline{\rm DR}}(M_Z)}{\sin\theta_{W,\rm susy}^{\overline{\rm DR}}(M_Z)}. \tag{19}$$

6.2.2. Calculation of the Yukawa couplings $y_f(M_Z)$

At the low-scale the $\overline{\rm DR}$ Yukawa coupling matrices $y_f^{\overline{\rm DR}}(M_Z)$ ($f = u, d, e$) in the SUSY model are calculated automatically if the user has set y_f to the value `Automatic` in the FlexibleSUSY model file. This is done for example in the CMSSM model file in Section 5. In this case FlexibleSUSY expresses the Yukawa couplings in terms of the fermion mass matrices m_u, m_d, m_e. In the MSSM, for example, these relations read in the SLHA convention [66]

$$y_u^{\overline{\rm DR}}(M_Z) = \frac{\sqrt{2}m_u^T}{v_u}, \qquad y_d^{\overline{\rm DR}}(M_Z) = \frac{\sqrt{2}m_d^T}{v_d}, \qquad y_e^{\overline{\rm DR}}(M_Z) = \frac{\sqrt{2}m_e^T}{v_d}, \tag{20}$$

where the superscript T means transposition of a matrix. The fermion mass matrices are composed as

$$m_u = \mathrm{diag}(m_u^{\rm input}, m_c^{\rm input}, m_{t,\rm susy}^{\overline{\rm DR}}(M_Z)), \tag{21}$$

$$m_d = \mathrm{diag}(m_d^{\rm input}, m_s^{\rm input}, m_{b,\rm susy}^{\overline{\rm DR}}(M_Z)), \tag{22}$$

$$m_e = \mathrm{diag}(m_e^{\rm input}, m_\mu^{\rm input}, m_{\tau,\rm susy}^{\overline{\rm DR}}(M_Z)), \tag{23}$$

where the values for $m_{u,c,d,s,e,\mu}^{\rm input}$ are read from the `SMINPUTS` block of the SLHA input file [65]. The CKM mixing matrix is currently assumed to be diagonal. The third generation quark masses are calculated in the $\overline{\rm DR}$ scheme from the SLHA user input quantities $m_t^{\rm pole}$, $m_{b,\rm SM}^{\overline{\rm MS}}(M_Z)$ and $m_{\tau,\rm SM}^{\overline{\rm MS}}(M_Z)$ [65]. In detail, the top quark $\overline{\rm DR}$ mass is calculated as

$$\begin{aligned} m_{t,\rm susy}^{\overline{\rm DR}}(\mu) = {} & m_t^{\rm pole} + \mathrm{Re}\,\Sigma_t^S(m_t^{\rm pole}) \\ & + m_t^{\rm pole}\left[\mathrm{Re}\,\Sigma_t^L(m_t^{\rm pole}) + \mathrm{Re}\,\Sigma_t^R(m_t^{\rm pole}) + \Delta m_t^{(1),\rm qcd} + \Delta m_t^{(2),\rm qcd}\right], \end{aligned} \tag{24}$$

where the Σ_t is the top one-loop self-energy without QCD contributions. The labels L, R, S denote the left-, right- and non-polarized part of the self-energy. The separated QCD corrections $\Delta m_t^{(1),\rm qcd}$

and $\Delta m_t^{(2),\mathrm{qcd}}$ are taken from [82] and read

$$\Delta m_t^{(1),\mathrm{qcd}} = -\frac{g_3^2}{12\pi^2}\left[5 - 3\log\left(\frac{m_t^2}{\mu^2}\right)\right], \tag{25}$$

$$\Delta m_t^{(2),\mathrm{qcd}} = \left(\Delta m_t^{(1),\mathrm{qcd}}\right)^2 - \frac{g_3^4}{4608\pi^4}\left[396\log^2\left(\frac{m_t^2}{\mu^2}\right) - 1476\log\left(\frac{m_t^2}{\mu^2}\right) - 48\zeta(3) + 2011 + 16\pi^2(1+\log 4)\right]. \tag{26}$$

The $\overline{\mathrm{DR}}$ mass of the bottom quark is calculated as [83, 65]

$$m_{b,\mathrm{susy}}^{\overline{\mathrm{DR}}}(\mu) = \frac{m_{b,\mathrm{SM}}^{\overline{\mathrm{DR}}}(\mu)}{1 - \mathrm{Re}\,\Sigma_b^{S,\mathrm{heavy}}(m_{b,\mathrm{SM}}^{\overline{\mathrm{MS}}})/m_b - \mathrm{Re}\,\Sigma_b^{L,\mathrm{heavy}}(m_{b,\mathrm{SM}}^{\overline{\mathrm{MS}}}) - \mathrm{Re}\,\Sigma_b^{R,\mathrm{heavy}}(m_{b,\mathrm{SM}}^{\overline{\mathrm{MS}}})}, \tag{27}$$

$$m_{b,\mathrm{SM}}^{\overline{\mathrm{DR}}}(\mu) = m_{b,\mathrm{SM}}^{\overline{\mathrm{MS}}}(\mu)\left(1 - \frac{\alpha_s}{3\pi} - \frac{23}{72}\frac{\alpha_s^2}{\pi^2} + \frac{3g_2^2}{128\pi^2} + \frac{13g_Y^2}{1152\pi^2}\right), \tag{28}$$

where $\tan\beta$ enhanced loop self-energy corrections are resummed. Finally, the $\overline{\mathrm{DR}}$ mass of the τ is calculated as

$$\begin{aligned} m_{\tau,\mathrm{susy}}^{\overline{\mathrm{DR}}}(\mu) &= m_{\tau,\mathrm{SM}}^{\overline{\mathrm{DR}}}(\mu) + \mathrm{Re}\,\Sigma_\tau^{S,\mathrm{heavy}}(m_{\tau,\mathrm{SM}}^{\overline{\mathrm{MS}}}) \\ &\quad + m_{\tau,\mathrm{SM}}^{\overline{\mathrm{DR}}}(\mu)\left[\mathrm{Re}\,\Sigma_\tau^{L,\mathrm{heavy}}(m_{\tau,\mathrm{SM}}^{\overline{\mathrm{MS}}}) + \mathrm{Re}\,\Sigma_\tau^{R,\mathrm{heavy}}(m_{\tau,\mathrm{SM}}^{\overline{\mathrm{MS}}})\right], \end{aligned} \tag{29}$$

$$m_{\tau,\mathrm{SM}}^{\overline{\mathrm{DR}}}(\mu) = m_{\tau,\mathrm{SM}}^{\overline{\mathrm{MS}}}(\mu)\left(1 - 3\frac{g_Y^2 - g_2^2}{128\pi^2}\right). \tag{30}$$

In the above equations $\Sigma_{b,\tau}^{\mathrm{heavy}}$ are the one-loop self-energies of the bottom and τ, where contributions from the gluon and photon are omitted. To convert the fermion masses from the $\overline{\mathrm{MS}}$ to the $\overline{\mathrm{DR}}$ scheme the Yukawa coupling conversion from [73] is used and it is assumed that the VEV is defined in the $\overline{\mathrm{DR}}$ scheme.

6.2.3. Electroweak symmetry breaking

FlexibleSUSY assumes that each SUSY model contains Higgs bosons, which trigger a spontaneous breaking of the electroweak symmetry. The corresponding EWSB consistency conditions are formulated in FlexibleSUSY at the one-loop level as

$$0 = \frac{\partial V^{\mathrm{tree}}}{\partial v_i} - t_i, \tag{31}$$

where V^{tree} is the tree-level Higgs potential, v_i is the VEV corresponding to the Higgs fields H_i and t_i is the one-loop tadpole diagram of H_i. The electroweak symmetry breaking conditions (31) are solved simultaneously using the iterative multi-dimensional root finder algorithm `gsl_multiroot_fsolver_hybrid` from the GNU Scientific Library (GSL). If no root can be found, the `gsl_multiroot_fsolver_hybrids` algorithm is tried as alternative, which uses a variable step size but might be a little slower.

In the CMSSM example from Section 5 the Eqs. (31) are expressed in the form of the following C++ function:

```
int MSSM<Two_scale>::tadpole_equations(const gsl_vector* x, void* params,
                                       gsl_vector* f)
{
   ...

   double tadpole[number_of_ewsb_equations];

   model->set_BMu(gsl_vector_get(x, 0));
   model->set_Mu(INPUT(SignMu) * Abs(gsl_vector_get(x, 1)));

   // calculate tree-level tadpole eqs.
   tadpole[0] = model->get_ewsb_eq_vd();
   tadpole[1] = model->get_ewsb_eq_vu();

   // subtract one-loop tadpoles
   if (ewsb_loop_order > 0) {
      model->calculate_DRbar_parameters();
      tadpole[0] -= Re(model->tadpole_hh(0));
      tadpole[1] -= Re(model->tadpole_hh(1));
   }

   for (std::size_t i = 0; i < number_of_ewsb_equations; ++i)
      gsl_vector_set(f, i, tadpole[i]);

   return GSL_SUCCESS;
}
```

The function parameter `x` is the vector of EWSB output parameters (defined in `EWSBOutputParameters`) and `f` is a vector which contains the one-loop EWSB Eqs. (31). This `tadpole_equations()` function is passed to the root finder, which searches for values of the model parameters μ and $B\mu$ until the Eqs. (31) are fulfilled.

If higher accuracy is required additional routines with higher order corrections can be added by setting `UseHiggs2LoopMSSM = True` in the model file. For example in the MSSM by default FlexibleSUSY adds two-loop Higgs FORTRAN routines supplied by P. Slavich from [60, 62] to add two-loop corrections of $O(\alpha_t\alpha_s)$, $O(\alpha_b\alpha_s)$, $O(\alpha_t^2)$, $O(\alpha_b^2)$, $O(\alpha_\tau^2)$ and $O(\alpha_t\alpha_b)$. In the NMSSM the same contributions can be added by setting `UseHiggs2LoopNMSSM = True` in the model file.

6.3. Tree-level spectrum

The tree-level $\overline{\text{DR}}$ masses are calculated from the $\overline{\text{DR}}$ model parameters by diagonalizing the mass matrices returned from `SARAH‘MassMatrix[]`. The numerical singular value decomposition is performed by the Eigen library routine `Eigen::JacobiSVD` for matrices with less than four rows and columns, and the LAPACK routines `zgesvd`, `dgesvd` for larger matrices. For the other types of diagonalization, `Eigen::SelfAdjointEigenSolver` from Eigen is used regardless of the matrix size.

FlexibleSUSY uses the following conventions for the diagonalization: A mass matrix M^2 for real scalar fields ϕ_i is diagonalized with an orthogonal matrix O as

$$\mathcal{L}_{m,\text{real scalar}} = -\frac{1}{2}\phi^T M^2 \phi = -\frac{1}{2}(\phi^m)^T M_D^2 \phi^m, \tag{32}$$

$$M^2 = (M^2)^T, \qquad \phi^m = O\phi, \qquad M_D^2 = O M^2 O^T, \qquad O^T O = \mathbf{1}, \tag{33}$$

where M_D^2 is diagonal and ϕ_i^m are the mass eigenstates. In case of complex scalar fields ϕ_i we use

$$\mathcal{L}_{m,\text{complex scalar}} = -\phi^\dagger M^2 \phi = -(\phi^m)^\dagger M_D^2 \phi^m, \tag{34}$$

$$M^2 = (M^2)^\dagger, \qquad \phi^m = U\phi, \qquad M_D^2 = U M^2 U^\dagger, \qquad U^\dagger U = \mathbf{1}. \tag{35}$$

A (possibly complex) symmetric mass matrix Y for Weyl spinors ψ_i is diagonalized as

$$\mathcal{L}^{\text{symm.}}_{m,\text{fermion}} = -\frac{1}{2}\psi^T Y \psi + \text{h.c.} = -\frac{1}{2}\chi^T Y_D \chi + \text{h.c.}, \tag{36}$$

$$Y = Y^T, \qquad Y_D = Z^* Y Z^\dagger, \qquad \chi = Z\psi, \qquad Z^\dagger Z = \mathbf{1}, \tag{37}$$

where Y_D is diagonal and χ_i are the mass eigenstates. The phases of Z are chosen such that all mass eigenvalues are positive. In case of a non-symmetric mass matrix X for Weyl spinors ψ_i we use

$$\mathcal{L}^{\text{svd}}_{m,\text{fermion}} = -(\psi^-)^T X \psi^+ + \text{h.c.} = -(\chi^-)^T X_D \chi^+ + \text{h.c.}, \tag{38}$$

$$\chi^+ = V\psi^+, \qquad \chi^- = U\psi^-, \qquad X_D = U^* X V^{-1}, \qquad U^\dagger U = \mathbf{1} = V^\dagger V, \tag{39}$$

where we're again choosing the phases of U and V such that all mass eigenvalues are positive.

6.4. Two-scale fixed point iteration

As explained at the beginning of Section 6, the RGEs plus the user-defined boundary conditions on the model parameters form a boundary value problem. FlexibleSUSY provides a default two-scale boundary value problem solver, which tries to find a set of model parameters consistent with all constraints at all scales. It does so by running iteratively between the scales of all boundary conditions, imposing the constraints (by calling the corresponding `apply()` function) and checking for convergence after each iteration. This approach is described in [84] originally for the MSSM and is widely implemented in SUSY spectrum generators. Despite sharing the same algorithm with others, the boundary value problem solver class from FlexibleSUSY, named `RGFlow`, has two notable properties. First, it extends the aforementioned procedure to towers of models. If the problem involves more than one model, `RGFlow` matches one model to the next after running the model parameters to the matching scale. Second, `RGFlow` is an abstract implementation of the algorithm, unaware of physics, in that it is free of hard-wired model-dependent code related to RGEs, boundary or matching conditions, or initial guesses. All these pieces of physics information are carried by separate objects which one then links to `RGFlow` to set up a boundary value problem. This modular design makes it easy to replace any of the above components, as shall be demonstrated in Section 7.2.2.

In more detail the two-scale algorithm used in FlexibleSUSY, as applied to a problem with a single MSSM-like model, works as follows, see also Figure 2:

Initial guess: The RG solver starts to guess all model parameters at the low-scale.

1. At the M_Z scale the gauge couplings $g_{1,2,3}$ are set to the known Standard Model values (ignoring threshold corrections).
2. The user-defined initial guess at the low-scale (defined in `InitialGuessAtLowScale`) is imposed. In the example given in Section 5 the Higgs VEVs are set to

$$v_d = v\cos\beta, \qquad v_u = v\sin\beta, \tag{40}$$

where $v = 246.22\,\text{GeV}$. Afterwards, the Yukawa couplings $y_{u,d,e}$ of the SUSY model are set from the known Standard Model Yukawa couplings using the tree-level relations (ignoring SUSY radiative corrections).

3. The SUSY parameters are run to the user-supplied first guess of the high-scale (`HighScaleFirstGuess`).
4. The high-scale boundary condition is imposed (defined in `HighScaleInput`). Afterwards, the user-defined initial guess for the remaining model parameters (defined in `InitialGuessAtHighScale`) is imposed. In the example given in Section 5 the superpotential parameter μ is set to the value 1.0 and its corresponding soft-breaking parameter $B\mu$ is set to zero.
5. All model parameters are run to the first guess of the low-scale (`LowScaleFirstGuess`).
6. The EWSB eqs. are solved at the tree-level.
7. The $\overline{\text{DR}}$ mass spectrum is calculated.

At this point all model parameters are set to some initial values and a first estimation of the $\overline{\text{DR}}$ mass spectrum is known. Now the actual iteration starts

Fixed-point iteration:

1. All model parameters are run to the low-scale (`LowScale`).
   (a) The $\overline{\text{DR}}$ mass spectrum is calculated.
   (b) The low-scale is recalculated. In the above example this step is trivial, because the low-scale is fixed to be M_Z.
   (c) The $\overline{\text{DR}}$ gauge couplings $g_{1,2,3}(M_Z)$ of the SUSY model are calculated using threshold corrections as described in Section 6.2.1.
   (d) The user-defined low-scale constraint is imposed (`LowScaleInput`). In the example above, the Yukawa couplings are calculated automatically as described in Section 6.2.2 and the Higgs VEVs are set to

$$v_d(M_Z) = \frac{2M_Z^{\overline{\text{DR}}}(M_Z)}{\sqrt{0.6g_1^2(M_Z) + g_2^2(M_Z)}\cos\beta(M_Z)}, \tag{41}$$

$$v_u(M_Z) = \frac{2M_Z^{\overline{\text{DR}}}(M_Z)}{\sqrt{0.6g_1^2(M_Z) + g_2^2(M_Z)}\sin\beta(M_Z)}. \tag{42}$$

   Since the Hypercharge gauge coupling g_1 is GUT normalized, the normalization factor $\sqrt{3/5}$ has to be included in the above relations.
2. Run all model parameters to the high-scale (`HighScale`).
   (a) Recalculate the high-scale as

$$M_X' = M_X \exp\left(\frac{g_2(M_X) - g_1(M_X)}{\beta_{g_1} - \beta_{g_2}}\right), \tag{43}$$

   where β_{g_i} is the two-loop β-function of the gauge coupling g_i. The value M_X' is used as new high-scale in the next iteration.
   (b) Impose the high-scale constraint (`HighScaleInput`). In the CMSSM example the following soft-breaking parameters are fixed to the universal values m_0, $M_{1/2}$ and A_0:

$$A^f(M_X) = A_0 \qquad (f = u, d, e), \tag{44}$$

$$m_{H_i}^2(M_X) = m_0^2 \qquad (i = 1, 2), \tag{45}$$

$$m_f^2(M_X) = m_0^2\mathbf{1} \qquad (f = q, l, d, u, e), \tag{46}$$

$$M_i(M_X) = M_{1/2} \qquad (i = 1, 2, 3). \tag{47}$$

3. Run model parameters to the SUSY-scale (`SUSYScale`).
   (a) Calculate the $\overline{\text{DR}}$ mass spectrum.
   (b) Recalculate the SUSY-scale M_S as

$$M_S = \sqrt{m_{\tilde{u}_1} m_{\tilde{u}_6}}, \tag{48}$$

   where $m_{\tilde{u}_1}$ and $m_{\tilde{u}_6}$ are the lightest and heaviest up-type squarks, respectively.
   (c) Impose the SUSY-scale constraint (`SUSYScaleInput`). In the example above, this step is trivial since `SUSYScaleInput` is set to be empty.
   (d) Solve the EWSB equations iteratively at the loop level. In the MSSM example from above leading two-loop corrections have been enabled by setting `UseHiggs2LoopMSSM = \ True`. This will add two-loop tadpole contributions to the effective Higgs potential during the EWSB iteration.
4. If not converged yet, goto 1. Otherwise, finish the iteration.

If the fixed-point iteration has converged, all $\overline{\text{DR}}$ model parameters are known at all scales between `LowScale` and `HighScale`. In this case all model parameters are run to the SUSY-scale and the pole-mass spectrum is calculated as described in Section 6.5. If the user has chosen a specific output scale for the running $\overline{\text{DR}}$ model parameters by setting entry 12 in block `MODSEL` in the SLHA input file, all model parameters are finally run to the defined output scale.

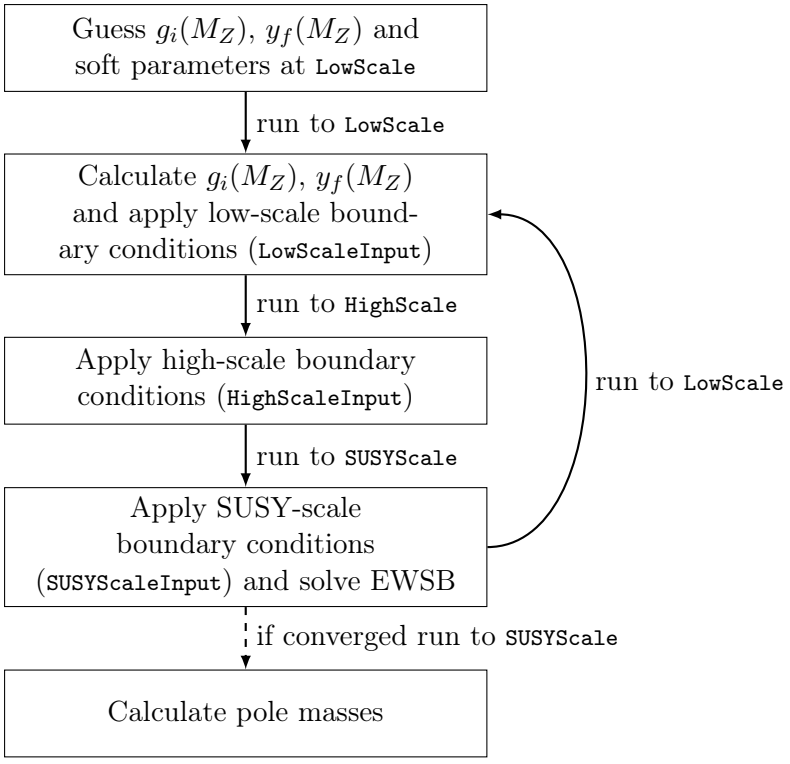

Figure 2: Iterative two-scale algorithm to calculate the spectrum.

During the fixed-point iteration several problems can appear. First of all, the iteration is not guaranteed to converge. If the desired accuracy goal is not achieved with the given maximum

number of iterations, FlexibleSUSY will set the `no_convergence` flag in the `Problems` class. This class monitors the problem status of the spectrum generator during the iteration and can be obtained from the model class via the `get_problems()` function. Besides non-convergence, solving the EWSB conditions (31) numerically with the desired accuracy might fail. In this case the `no_ewsb` flag is set. Furthermore, in intermediate iteration steps tachyonic states might appear, which are ignored but nevertheless monitored in the `Problems` class. If tachyons still exist after the iteration has converged the mass spectrum is marked as invalid by setting entry 4 in the `SPINFO` block in the SLHA output file. Finally, during the RG running some couplings might become non-perturbative. In this case the iteration stops setting the `no_perturbative` flag.

It is important to note that in the case of such problem points it is non-trivial to judge whether this is because there is no physical solution for the given parameter space point or whether a solution exists but the fixed point iteration is unable to find the solution. While FlexibleSUSY makes it as easy as possible to find spectra, when studying new models a physical understanding of the model is still essential and this can help the user determine why the such problems arise.

Nonetheless FlexibleSUSY provides help for such cases in several ways. One may adjust initial guesses specified in the FlexibleSUSY model file, such as changing the choice of `HighScaleFirstGuess` or altering `HighScaleMinimum` and `HighScaleMaximum` which can be used to push the iteration back towards where the solution should be if it gets off track. For experienced users the clear code structure also allows the possibility of direct adaption of the code.

Finally instead of tinkering with the two-scale solver one may wish to replace it entirely. The modular design of FlexibleSUSY allows for the solver for the boundary value problem to be replaced. An alternative solver with potentially better convergence properties (at the expense of slower speed) is already planned for a later release.

6.5. Pole masses

After the solver routine has finished and convergence has been achieved, all $\overline{\text{DR}}$ parameters consistent with the EWSB conditions, low energy data and all user-supplied boundary conditions are known at any scale between `LowScale` and `HighScale`.

The (physical) pole mass spectrum can now be calculated. FlexibleSUSY uses the full one-loop self-energies and tree-level mass matrices obtained from SARAH to calculate the pole masses, which means finding the values p that solve the equation

$$0 = \det\left[p^2\mathbf{1} - m_{f,1L}(p^2)\right]. \tag{49}$$

Here the one-loop mass matrix $m_{f,1L}(p^2)$ for field f is given in terms of the tree-level mass matrix m_f and the self-energy $\Sigma_f(p^2)$ as

$$\text{scalars } \phi: \quad m_{\phi,1L}(p^2) = m_\phi - \Sigma_\phi(p^2), \tag{50}$$

$$\text{Majorana fermions } \chi: \quad m_{\chi,1L}(p^2) = m_\chi - \frac{1}{2}\Big[\Sigma^S_\chi(p^2) + \Sigma^{S,T}_\chi(p^2) + \Big(\Sigma^{L,T}_\chi(p^2) + \Sigma^R_\chi(p^2)\Big)m_\chi + m_\chi\Big(\Sigma^L_\chi(p^2) + \Sigma^{R,T}_\chi(p^2)\Big)\Big], \tag{51}$$

$$\text{Dirac fermions } \psi: \quad m_{\psi,1L}(p^2) = m_\psi - \Sigma^S_\psi(p^2) - \Sigma^R_\psi(p^2)m_\psi - m_\psi\Sigma^L_\psi(p^2). \tag{52}$$

Eq. (49) can be solved by diagonalizing the one-loop mass matrix $m_{f,1L}(p^2)$. However, since $m_{f,1L}(p^2)$ depends on the momentum p, an iteration over p must be performed. Since this iteration

can be very time consuming for large field multiplets, FlexibleSUSY provides two approximative procedures with a shorter run-time. The used procedure can be set in the model file for each field. These approximations work as follows:

- `LowPoleMassPrecision`: This option provides the lowest precision but is also the fastest one. Here the one-loop mass matrix $m_{f,1L}^{\text{low}}$ is calculated exactly once as

$$\forall i,j : (m_{f,1L}^{\text{low}})_{ij} = (m_{f,1L}(p^2 = m_{f_i} m_{f_j}))_{ij}, \tag{53}$$

  where m_{f_i} is the ith mass eigenvalue of the tree-level mass matrix m_f. Afterwards, $m_{f,1L}^{\text{low}}$ is diagonalized and the eigenvalues are interpreted as pole masses $m_{f_i}^{\text{pole}}$. This method neglects what are formally of two-loop order. This method is imprecise if the self-energy corrections to the mass matrix are large or the tree-level mass spectrum of the multiplet is very split.

- `MediumPoleMassPrecision` (default): This option provides calculation with medium precision with a medium execution time. Here the one-loop mass matrix $m_{f,1L}^{\text{medium}}$ is calculated n times as

$$(m_{f,1L}^{\text{medium}})_{ij}^{(k)} = (m_{f,1L}(p^2 = m_{f_k}^2))_{ij}, \qquad k = 1, \ldots, n, \tag{54}$$

  where m_{f_k} is the kth mass eigenvalue of the tree-level mass matrix m_f. Afterwards, each mass matrix $(m_{f,1L}^{\text{medium}})^{(k)}$ is diagonalized and the kth eigenvalue is interpreted as pole mass $m_{f_k}^{\text{pole}}$. This method is imprecise if the self-energy corrections to the mass matrix are large.

- `HighPoleMassPrecision`: This option solves Eq. (49) exactly by iterating over the momentum p. It therefore provides the determination of the pole masses with highest precision, but has also the highest execution time. Here the one-loop mass matrix $m_{f,1L}^{\text{high}}$ is diagonalized n times, as in the case of `MediumPoleMassPrecision`, resulting in n pole masses $m_{f_k}^{\text{pole}}$ $(k = 1, \ldots, n)$. Afterwards, the diagonalization is repeated, this time using the calculated pole masses $m_{f_k}^{\text{pole}}$ for the momentum calculation $p^2 = (m_{f_k}^{\text{pole}})^2$. The iteration stops if convergence is reached.

If higher accuracy is required, two-loop corrections to the self-energies can be added by setting `UseHiggs2LoopMSSM = True` in the MSSM or `UseHiggs2LoopNMSSM = True` in the NMSSM in the model file. This enables two-loop Higgs FORTRAN routines supplied by P. Slavich from [58, 59, 60, 61, 62] which add two-loop corrections of $O(\alpha_t\alpha_s)$, $O(\alpha_b\alpha_s)$, $O(\alpha_t^2)$, $O(\alpha_b^2)$, $O(\alpha_\tau^2)$ and $O(\alpha_t\alpha_b)$. Something similar is done for the NMSSM, but in this case the NMSSM $O(\alpha_t\alpha_s)$, $O(\alpha_b\alpha_s)$ pieces come from [63], while for $O(\alpha_t^2)$, $O(\alpha_b^2)$, $O(\alpha_\tau^2)$ and $O(\alpha_t\alpha_b)$ the MSSM pieces are used though it should be understood that these are not complete in the NMSSM. For other models since the Higgs mass is a very important measurement and the two-loop corrections can be larger than the current experimental error [63] we recommend that the leading log two-loop corrections are estimated, by generalizing those of the MSSM or NMSSM, as has been done, for example, in the E_6SSM [33].

7. Flexible Applications

By definition, research is an endeavor to find something new. Therefore, it can often be the case that a spectrum generator right out of the box is not enough. FlexibleSUSY attempts to offer a clean interface through which one can exploit its facilities while undergoing a minimal amount of

frustration, when one programs for a wide variety of studies. We provide two basic levels for the user to create a custom spectrum generator: (i) The Mathematica level, where one writes or adapts a model file and (ii) the C++ level, where the generated classes can be extended, recombined or replaced by self-made modules. In what follows, adaptions on these two levels shall be demonstrated by presenting a few use cases at differing degrees of complexity.

To avoid confusion, it should be mentioned that the code snippets presented below are not verbatim listings of the files included in the package. They have been tailored retaining the semantics for conciseness.

7.1. Adapting model files

There are simple but interesting goals that one can achieve only by working on Mathematica files. The outcome thus obtained from FlexibleSUSY might already include a fully-fledged program that is useful in physics analysis. In a more advanced project, one might utilize the produced libraries as building blocks that constitute the target application. For a general account of the FlexibleSUSY model files, refer to Section 5.

7.1.1. Changing boundary conditions

As already emphasized in Section 3, the modular design of FlexibleSUSY makes it straightforward to replace a boundary condition object. The question then becomes how one could obtain an alternative boundary condition class, apart from writing one by hand. The meta code feature of FlexibleSUSY offers great assistance in this respect. An example shall be presented to illustrate how this works.

In the literature, there is a popular alternative to the CMSSM boundary condition under which the Higgs soft masses are allowed to be different from the universal mass of the other scalars [85]. One might implement this non-universal Higgs-mass MSSM (NUHMSSM) scenario simply by modifying the model description given to FlexibleSUSY. A section of the `FlexibleSUSY.m.in` file is listed below:

```
1 EXTPAR = {{1, mHd2In}, {2, mHu2In}};
2
3 HighScaleInput={
4   {mHd2, mHd2In}, {mHu2, mHu2In},
5   {T[Ye], Azero*Ye}, {T[Yd], Azero*Yd}, {T[Yu], Azero*Yu},
6   {mq2, UNITMATRIX[3] m0^2}, {ml2, UNITMATRIX[3] m0^2}, {md2, UNITMATRIX[3] m0^2},
7   {mu2, UNITMATRIX[3] m0^2}, {me2, UNITMATRIX[3] m0^2},
8   {MassB, m12}, {MassWB, m12}, {MassG, m12}
9 };
```

Since `mHd2` and `mHu2` are to be fixed at constants different from `m0^2`, two additional input parameters, `mHd2In` and `mHu2In`, holding those constants, are introduced in the list `EXTPAR`. These input parameters are then declared to be the high-scale values of `mHd2` and `mHu2` in line 4. The rest of the boundary conditions is the same as in the CMSSM. In the SLHA input file, the parameter indices `1` and `2` of `mHd2In` and `mHu2In`, declared in `EXTPAR` above, must appear as the first field in each line in the `EXTPAR` block:

```
1 Block EXTPAR
2     1    10000                # mHd2In
```

```
3     2    -2500                # mHu2In
```

Note that the two additional input parameters are chosen to have mass dimension 2, unlike `m0`. This makes it easy to try both signs of the high-scale value of either soft Higgs mass squared, as exemplified in line 3. If one were not interested in a negative boundary value of `mHu2` for instance, then a dimension-1 parameter might instead be introduced whose square is equated with `mHu2`.

The full implementation is available in `model_files/NUHMSSM/`. To try it out, do the following:

```
1 $ ./createmodel --name=NUHMSSM --sarah-model=MSSM
2 $ ./configure --with-models=NUHMSSM
3 $ make
4 $ models/NUHMSSM/run_NUHMSSM.x \
    --slha-input-file=models/NUHMSSM/LesHouches.in.NUHMSSM
```

Notice the `--sarah-model=MSSM` flag in line 1. It tells the `createmodel` script to reuse the MSSM specification in SARAH to generate the C++ program.

7.1.2. Extending existing models

The preceding example was a modest alteration of a physics scenario in that an existing model has been reused. A more non-trivial modification might involve an extension of the particle content as well as the interactions. One of the simplest classes of models beyond the MSSM is those with additional gauge-singlet fields. In what follows, a supersymmetric type-I see-saw model [86] shall be considered in which three neutral (heavy) chiral superfields are introduced.

In the package, this model is named `MSSMRHN`, standing for the MSSM plus right-handed neutrinos. One needs to prepare an input file to SARAH which might be placed in `<FlexibleSUSY-root>/sarah/MSSMRHN/` or `<SARAH-root>/Models/MSSMRHN/`. The input file `MSSMRHN.m` contains the declaration of the three-generation singlets `v`:

```
1 SuperFields[[8]] = {v, 3, conj[vR], 0, 1, 1, RpM};
```

as well as the neutrino Yukawa couplings and the Majorana mass terms of the singlets:

```
2 SuperPotential = Yu u.q.Hu - Yd d.q.Hd - Ye e.l.Hd + \[Mu] Hu.Hd +
3                  Yv v.l.Hu + Mv/2 v.v;
```

Further declarations inform SARAH of how to form Dirac spinors out of the new Weyl spinors and how the scalars and the fermions mix to comprise the mass eigenstates:

```
4  DEFINITION[GaugeES][DiracSpinors] = {
5    Fu1 -> {FuL, 0}, Fu2 -> {0, FuR},
6    Fv1 -> {FvL, 0}, Fv2 -> {0, FvR},
7    ...
8  };
9
10 DEFINITION[EWSB][MatterSector] = {
11   {{SuL, SuR}, {Su, ZU}},
12   {{SvL, SvR}, {Sv, ZV}},
13   ...
```

```
14    {{fB, fW0, FHd0, FHu0}, {L0, ZN}},
15    {{FvL, conj[FvR]}, {FV, UV}},
16    {{{fWm, FHdm}, {fWp, FHup}}, {{Lm, UM}, {Lp, UP}}},
17    {{{FuL}, {conj[FuR]}}, {{FUL, ZUL}, {FUR, ZUR}}}
18  };
19
20  DEFINITION[EWSB][DiracSpinors] = {
21    Fu  -> {FUL, conj[FUR]},
22    Fv  -> {FV , conj[FV] },
23    Chi -> {L0 , conj[L0] },
24    Cha -> {Lm , conj[Lp] },
25    ...
26  };
```

With respect to the MSSM file, the newly added lines are 6, 12, 15, and 22. Notice that the (left- and right-handed) neutrino mixing in line 15 resembles the neutralino mixing in line 14. Due to the Majorana mass term in the superpotential, the six neutrino mass eigenstates are described in terms of Majorana spinors like the neutralinos.

One should then add descriptions of the new states in the file `particles.m`:

```
1   ParticleDefinitions[GaugeES] = {
2     {Fv1, { Description -> "Dirac Left Neutrino" }},
3     {Fv2, { Description -> "Dirac Right Neutrino" }},
4     {SvR, { Description -> "Right Sneutrino", LaTeX ->"\\tilde{\\nu}_R"}},
5     ...
6   };
7
8   ParticleDefinitions[EWSB] = {
9     {Sv, { Description -> "Sneutrinos",
10           PDG -> {1000012, 1000014, 1000016, 2000012, 2000014, 2000016}}},
11    {Fv, { Description -> "Neutrinos",
12           PDG -> {12, 14, 16, 9900012, 9900014, 9900016}}},
13    ...
14  };
15
16  WeylFermionAndIndermediate = {
17    {v,   { Description -> "Right Neutrino Superfield" }},
18    {FV,  { Description -> "Neutrino-Masseigenstate"}},
19    {FvL, { Description -> "Left Neutrino"}},
20    {FvR, { Description -> "Right Neutrino"}},
21    ...
22  };
```

In line 12, one finds PDG codes beginning with `99`. Such numbers are available for a program author's private use [80]. The new parameters in the superpotential and the soft supersymmetry breaking sector are to be described in `parameters.m`:

```
1   ParameterDefinitions = {
2     {UV,    { Description -> "Neutrino-Mixing-Matrix"}},
3     {Yv,    { Description -> "Neutrino-Yukawa-Coupling" }},
4     {T[Yv], { Description -> "Trilinear-Neutrino-Coupling"}},
5     {Mv,    { LaTeX -> "M_v", OutputName -> Mv, LesHouches -> Mv}},
6     {B[Mv], { LaTeX -> "B_v", OutputName -> BMv, LesHouches -> BMv}},
```

```
7   {mv2,   { Description -> "Softbreaking right Sneutrino Mass"}},
8   ...
9 };
```

For further details on how to write model files for SARAH, we refer to its manual [69, 57].

Finally, it remains to put `FlexibleSUSY.m.in` in `model_files/MSSMRHN/`. The high-scale boundary conditions therein might read:

```
1 HighScaleInput = {
2   {Mv, LHInput[Mv]}, {B[Mv], LHInput[B[Mv]]},
3   {Yv, LHInput[Yv]}, {T[Yv], Azero*Yv},
4   {mv2, UNITMATRIX[3] m0^2},
5   ...
6 };
```

In this particular example, the new parameters shown in lines 2–3 are constrained at the M_X scale to the values given in the SLHA input file. In a data-driven approach, they might alternatively be fixed from a matching against a low-energy theory containing the dimension-5 neutrino mass operator in conjunction with supplementary constraints.

With the above set of input files, FlexibleSUSY can generate the C++ code and compile it to produce `libMSSMRHN`. These products shall be employed as the implementation of the theory that underlies the MSSM in the next subsection.

7.2. Adapting C++ code

There are problems which one cannot solve only by editing Mathematica model files. To unlock the full potential of FlexibleSUSY, it is an advantage not to avoid programming at the C++ level. For this, it should help to have working knowledge about the basic structure of a spectrum generator, set out in Section 6. In what follows, it shall be demonstrated that the clean class structure serves as firm guidance on the job.

7.2.1. Stacking models in a tower of effective theories

Consider a physics scenario which is best described by a tower of effective theories. Within the framework of FlexibleSUSY, the C++ class structure is a faithful reflection of this physicist's view on the given problem. Here we illustrate this point using a well-known configuration in which the higher-energy theory is the MSSMRHN discussed above which gives rise to the MSSM as the lower-energy effective theory. The relevant classes are sketched in Figure 3. The `MSSMRHN` object is in effect from the M_X scale down to the M_ν scale at which the right-handed neutrinos are decoupled. Below this scale, the `MSSM` object takes over. On the left of the vertical axis, the boundary condition objects acting on either model are displayed, together with the matching object connecting the two theories. Note that each of the boundary condition and matching objects maintains and updates its own scale over iterations. An arrow in the figure depicts the association of a constraint with its scale. All these components are plugged into the `RGFlow` object which then solves the problem.

The matching class as well as gluing codes have to be written by hand to build such a program.[2] All remaining components of a multi-model spectrum generator can be authored by making a

[2] It is planned that a future release of FlexibleSUSY will be capable of creating this code automatically.

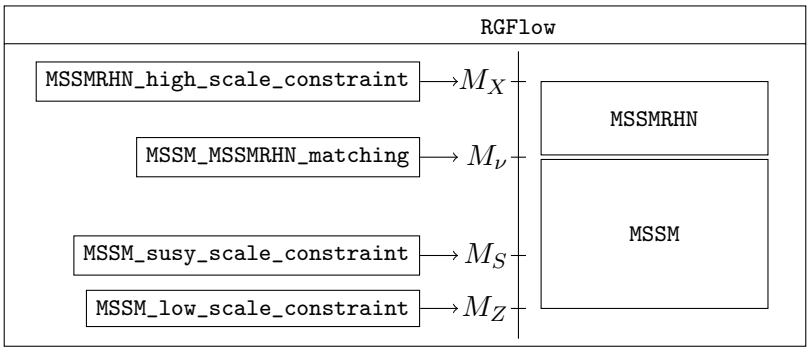

Figure 3: Schematic class structure in the C++ code for the tower scenario.

straightforward extension to each corresponding single-model counterpart for one of the models forming the tower.

As the target spectrum generator depends on two models, one should first build these prerequisites by:

```
1 $ ./createmodel --name=MSSM
2 $ ./createmodel --name=MSSMRHN
3 $ ./configure --with-models=MSSM,MSSMRHN
4 $ make
```

As a by-product, line 3 also creates a `Makefile` in `examples/tower/`. One can best see the overall code structure of the application in this file:

```
1  CPPFLAGS  := -I. $(INCCONFIG) $(INCFLEXI) $(INCLEGACY) $(INCSLHAEA) \
2               $(INCMSSM) $(INCMSSMRHN)
3
4  TOWER_SRC := run_tower.cpp \
5               MSSM_MSSMRHN_two_scale_matching.cpp \
6               MSSM_MSSMRHN_two_scale_initial_guesser.cpp
7
8  TOWER_OBJ := $(patsubst %.cpp, %.o, $(filter %.cpp, $(TOWER_SRC)))
9
10 run_tower.x: $(TOWER_OBJ) $(LIBMSSM) $(LIBMSSMRHN) $(LIBFLEXI) $(LIBLEGACY)
11   $(CXX) -o $@ $^ $(LOOPFUNCLIBS) $(GSLLIBS) $(BOOSTTHREADLIBS) $(THREADLIBS) \
     $(LAPACKLIBS) $(FLIBS)
```

The include directives in line 2 tell the compiler where to find the headers for either `MSSM` or `MSSMRHN`. The `.cpp` files in lines 4–6 and the `.hpp` files that they include are to be written by hand. Obviously, the executable `run_tower.x`, in line 10, depends on both `$(LIBMSSM)` and `$(LIBMSSMRHN)` that implement the auto-generated components in Figure 3.

To prepare the main source file `run_tower.cpp`, one can extend `run_MSSM.cpp` or `run_MSSMRHN.cpp` produced in either model directory. The shipped example reads:

```
1 #include "MSSM_MSSMRHN_spectrum_generator.hpp"
2
3 int main(int argc, char* argv[])
```

```
4  {
5    // define objects;
6    QedQcd oneset;
7    MSSM_input_parameters    input_1;
8    MSSMRHN_input_parameters input_2;
9    // fill in input_1 and input_2;
10   oneset.toMz(); // run SM fermion masses to MZ
11   typedef Two_scale algorithm_type;
12   MSSM_MSSMRHN_spectrum_generator<algorithm_type> spectrum_generator;
13   // set up spectrum_generator;
14   spectrum_generator.run(oneset, input_1, input_2);
15   // extract outcome from models;
16 }
```

where a line in the form `// ...;` shall be understood to be a pseudo-code. Given two models, one declares two sets of input parameters, `input_1` and `input_2`, in lines 7–8.

The crucial point is the definition of the `MSSM_MSSMRHN_spectrum_generator` object in line 12, which creates and drives the `RGFlow` object in Figure 3. This task is started by calling the `run()` member function in line 14. It is defined in `MSSM_MSSMRHN_spectrum_generator.hpp` and reads:

```
1  template<class T> void MSSM_MSSMRHN_spectrum_generator<T>::run
2  (const QedQcd& oneset,
3   const MSSM_input_parameters& input_1, const MSSMRHN_input_parameters& input_2)
4  {
5    high_scale_constraint_2.clear(); // of type MSSMRHN_high_scale_constraint<T>
6    susy_scale_constraint_1.clear(); // of type MSSM_susy_scale_constraint<T>
7    low_scale_constraint_1 .clear(); // of type MSSM_low_scale_constraint<T>
8    matching.reset();                // of type MSSM_MSSMRHN_matching<T>
9    high_scale_constraint_2.set_input_parameters(input_2);
10   susy_scale_constraint_1.set_input_parameters(input_1);
11   low_scale_constraint_1 .set_input_parameters(input_1);
12   matching.set_upper_input_parameters(input_2);
13   high_scale_constraint_2.initialize();
14   susy_scale_constraint_1.initialize();
15   low_scale_constraint_1 .initialize();
16   if (!is_zero(input_scale_2)) high_scale_constraint_2.set_scale(input_scale_2);
```

This piece of code is nearly a verbatim copy of the corresponding part of `MSSM_spectrum_generator.hpp`. The only differences are that the type of `high_scale_constraint_2` is `MSSMRHN_high_scale_constraint<T>` and that the `matching` object has been added. Recall that the template parameter `T` has been bound to `Two_scale` in the `main` function. One then constructs a list of the constraints on `MSSM`:

```
17   std::vector<Constraint<T>*> upward_constraints_1;
18   upward_constraints_1.push_back(&low_scale_constraint_1);
19   std::vector<Constraint<T>*> downward_constraints_1;
20   downward_constraints_1.push_back(&susy_scale_constraint_1);
21   downward_constraints_1.push_back(&low_scale_constraint_1);
```

and initializes the `MSSM` object:

```
22   model_1.clear();                 // of type MSSM<T>
```

```
23  model_1.set_input_parameters(input_1);
24  model_1.do_calculate_sm_pole_masses(calculate_sm_masses);
```

Likewise for `MSSMRHN`:

```
25  std::vector<Constraint<T>*> upward_constraints_2;
26  upward_constraints_2.push_back(&high_scale_constraint_2);
27  std::vector<Constraint<T>*> downward_constraints_2;
28  downward_constraints_2.push_back(&high_scale_constraint_2);
29  model_2.clear();                  // of type MSSMRHN<T>
30  model_2.set_input_parameters(input_2);
```

Note that `model_2` does not have to calculate the pole masses of the SM particles since it is active only above M_ν which is assumed to be much higher than the weak scale. To test the convergence of both models, one may construct a composite convergence tester out of auto-generated `MSSM_convergence_tester` and `MSSMRHN_convergence_tester`:

```
31  MSSM_convergence_tester<T> convergence_tester_1(&model_1, precision_goal);
32  MSSMRHN_convergence_tester<T> convergence_tester_2(&model_2, precision_goal);
33  if (max_iterations > 0) {
34    convergence_tester_1.set_max_iterations(max_iterations);
35    convergence_tester_2.set_max_iterations(max_iterations);
36  }
37  Composite_convergence_tester<T> convergence_tester;
38  convergence_tester.add_convergence_tester(&convergence_tester_1);
39  convergence_tester.add_convergence_tester(&convergence_tester_2);
```

On construction, the initial guesser accepts the following parameters including the two model objects:

```
40  MSSM_MSSMRHN_initial_guesser<T> initial_guesser
41    (&model_1, &model_2, input_2, oneset,
42     low_scale_constraint_1, susy_scale_constraint_1, high_scale_constraint_2,
43     matching);
```

The code of the above class shall be presented later on. One then passes `convergence_tester` and `initial_guesser` to `solver`, the `RGFlow` object, along with the precision specification:

```
44  Two_scale_increasing_precision precision(10.0, precision_goal);
45  solver.reset();                   // of type RGFlow<T>
46  solver.set_convergence_tester(&convergence_tester);
47  solver.set_running_precision(&precision);
48  solver.set_initial_guesser(&initial_guesser);
```

Finally, one is ready to construct the tower of effective theories by adding to `solver` each model plus the associated list of constraints optionally accompanied by a matching object:

```
49  solver.add_model(&model_1, &matching, upward_constraints_1, \
    downward_constraints_1);
50  solver.add_model(&model_2, upward_constraints_2, downward_constraints_2);
```

The order of addition is from the lowest scale to the highest. Notice in line 49 that the matching object between `model_1` and `model_2` is given when one adds the former, i.e. the lower-energy model. It then remains to solve the boundary value problem:

```
51   high_scale_2 = susy_scale_1 = low_scale_1 = 0; matching_scale = 0;
52   solver.solve();
```

After the solution is found, one can obtain the resulting low-energy spectrum. Since `model_1` is in contact with the lowest energy, let it calculate the spectrum:

```
53   susy_scale_1 = susy_scale_constraint_1.get_scale();
54   model_1.run_to(susy_scale_1);    // of type MSSM<T>
55   model_1.calculate_spectrum();
56   if (!is_zero(parameter_output_scale_1))
57     model_1.run_to(parameter_output_scale_1);
58 }
```

In lines 56–57, the scale is optionally brought to the value at which one wishes to get the $\overline{\text{DR}}$ parameters.

One needs to write the matching class for a particular pair of models from scratch. It shall be based on the abstract class `Matching<Two_scale>` that comes with FlexibleSUSY. In the present example, the class is declared in the header `MSSM_MSSMRHN_two_scale_matching.hpp`:

```
1  template<> class MSSM_MSSMRHN_matching<Two_scale> : public Matching<Two_scale> {
2  public:
3    MSSM_MSSMRHN_matching();
4    MSSM_MSSMRHN_matching(const MSSMRHN_input_parameters&);
5    void match_low_to_high_scale_model();
6    void match_high_to_low_scale_model();
7    double get_scale() const;
8    void set_models(Two_scale_model *lower, Two_scale_model *upper);
9    double get_initial_scale_guess() const;
10   void set_upper_input_parameters(const MSSMRHN_input_parameters&);
11   void set_scale(double);
12   void reset();
13 private:
14   MSSM   <Two_scale> *lower;
15   MSSMRHN<Two_scale> *upper;
16   void make_initial_scale_guess();
17   void update_scale();
18   ...
19 };
```

As lines 4 and 10 indicate, this class takes an `MSSMRHN_input_parameters` object as input. The `MvInput` field thereof is referenced for the initial guess of the matching scale in line 25 of `MSSM_MSSMRHN_two_scale_matching.cpp`:

```
1  void MSSM_MSSMRHN_matching<Two_scale>::match_low_to_high_scale_model()
2  {
3    upper->set_Yd(lower->get_Yd());
```

```
4     // copy rest of couplings from lower to upper;
5     upper->set_scale(lower->get_scale());
6   }
7
8   void MSSM_MSSMRHN_matching<Two_scale>::match_high_to_low_scale_model()
9   {
10    update_scale();
11    lower->set_Yd(upper->get_Yd());
12    // copy rest of couplings from upper to lower;
13    lower->set_scale(upper->get_scale());
14  }
15
16  void MSSM_MSSMRHN_matching<Two_scale>::set_upper_input_parameters
17  (const MSSMRHN_input_parameters& inputPars_)
18  {
19    inputPars = inputPars_;
20    make_initial_scale_guess();
21  }
22
23  void MSSM_MSSMRHN_matching<Two_scale>::make_initial_scale_guess()
24  {
25    double RHN_scale = pow(abs(inputPars.MvInput.determinant()), 1.0/3);
26    scale = initial_scale_guess = RHN_scale;
27  }
28
29  void MSSM_MSSMRHN_matching<Two_scale>::update_scale()
30  {
31    double RHN_scale = pow(abs(upper->get_Mv().determinant()), 1.0/3);
32    scale = RHN_scale;
33  }
```

Recall that `MvInput` is the high-scale boundary value of `Mv`. If one had opted for an alternative strategy to fix `Mv`, then `MSSM_MSSMRHN_matching` might have required a different input. Subsequently, the matching scale is updated at each iteration to be the geometric mean of the running `Mv` eigenvalues in lines 31–32. The actual matching process takes place in the two functions `match_low_to_high_scale_model` and `match_high_to_low_scale_model`. For brevity, their examples shown above simply copy each coupling upwards or downwards, neglecting threshold corrections. The way to incorporate these corrections should be self-evident from the code structure.

The last missing piece is the initial guesser. One can extend the already available `MSSM_initial_guesser` class. The essential task is done by the following member function:

```
1   void MSSM_MSSMRHN_initial_guesser<Two_scale>::guess()
2   {
3     // guess SUSY couplings in model-1 at low energy;
4
5     const double low_scale_guess_1 = low_constraint_1.get_initial_scale_guess();
6     const double high_scale_guess_2 = high_constraint_2.get_initial_scale_guess();
7     const double matching_scale_guess = matching.get_initial_scale_guess();
```

Compared to the MSSM case, the differences are that the type of `high_constraint_2` is `MSSMRHN_high_scale_constraint<Two_scale>` and that `matching_scale_guess` has been inserted. Due to this intermediate scale, the initial run-up is divided into two steps, with a matching procedure

in-between:

```
  model_1->run_to(matching_scale_guess); // of type MSSM<Two_scale>
  matching.set_models(model_1, model_2);
  matching.match_low_to_high_scale_model();
  model_2->run_to(high_scale_guess_2);   // of type MSSMRHN<Two_scale>
```

The high-scale constraints are applied to `model_2`, the higher-energy model, and the remaining undetermined parameters are guessed:

```
  high_constraint_2.set_model(model_2);
  high_constraint_2.apply();
  model_2->set_Mu(1.0); model_2->set_BMu(0.0);
```

The initial two-step run-down again involves a matching process:

```
  model_2->run_to(matching_scale_guess);
  matching.match_high_to_low_scale_model();
  model_1->run_to(low_scale_guess_1);
```

At the low scale where `MSSM` is valid, the code is the same as in `MSSM_initial_guesser`:

```
  model_1->solve_ewsb_tree_level();
  model_1->calculate_DRbar_parameters();
  model_1->set_thresholds(3); model_1->set_loops(2);
}
```

Finally, one prescribes the additional input parameters in the SLHA input file:

```
Block YvIN                    # neutrino Yukawas at MX
1 1   0.1568611               # Yv(1,1)
1 2   0.6400513               # Yv(1,2)
1 3   0.7521494               # Yv(1,3)
2 1   0.4663838               # Yv(2,1)
...                           # remaining 5 entries
Block MvIN                    # heavy neutrino masses at MX
1 1   4.150000E+14            # Mv(1,1)
...                           # remaining 8 entries
Block BMvIN                   # right-handed sneutrino bilinear terms
1 1   1.000000E+02            # BMv(1,1)
...                           # remaining 8 entries
```

The file in the package contains the values that approximately reproduce the observed neutrino masses and mixing angles [80] through the see-saw mechanism.

For further details, browse the directory `examples/tower/`. One can build and run the example therein by:

```
$ cd examples/tower
$ make
$ ./run_tower.x --slha-input-file=LesHouches.in.tower
```

In the output, a part of significant physical interest might be:

```
Block MSL2 Q=   8.97114431E+02
  1  1     1.25652698E+05   # ml2(1,1)
  1  2    -7.64196328E+01   # ml2(1,2)
  1  3    -7.08429254E+01   # ml2(1,3)
  2  1    -7.64196328E+01   # ml2(2,1)
  2  2     1.25388732E+05   # ml2(2,2)
  2  3    -3.15865242E+02   # ml2(2,3)
  3  1    -7.08429254E+01   # ml2(3,1)
  3  2    -3.15865242E+02   # ml2(3,2)
  3  3     1.24556532E+05   # ml2(3,3)
```

This result demonstrates the well-known effect on the off-diagonal slepton mass matrix elements from a non-trivial flavour structure of `Yv` [87]. This leads in turn to the slepton mixing matrices, `ZE` and `ZV`, which contain inter-generational mixings apart from the generic left-right mixings.

7.2.2. Integrating custom-built C++ components

In Section 7.1.1, it was explained how one can let FlexibleSUSY generate an alternative boundary condition class by authoring a model file. Nonetheless, the way to employ this class at the C++ level might still remain obscure to the reader since FlexibleSUSY automatically took care of it. Here, an example shall be exhibited with the emphasis on the modular C++ code structure that helps such programming tasks. Concretely, the auto-generated low-energy boundary condition on the MSSM shall be modified so that $\alpha_{\mathrm{s,susy}}^{\overline{\mathrm{DR}}}$ is determined from $\alpha_{\mathrm{s,SM}}^{(5),\overline{\mathrm{MS}}}$ by means of a two-loop matching. This shall be accompanied by an improvement of the g_3 β-function to the three-loop accuracy.

The first step is to alter the model class which evaluates the β-functions. Thanks to the `beta()` method being virtual, one can override it conveniently by deriving a class from `MSSM`. The declaration might look like:

```
1  #include "MSSM_two_scale_model.hpp"
2
3  template<>
4  class MSSMcbs<Two_scale> : public MSSM<Two_scale> {
5  public:
6    explicit MSSMcbs(const MSSM_input_parameters& input_ = MSSM_input_parameters());
7    virtual ~MSSMcbs();
8    virtual Eigen::ArrayXd beta() const;
9    MSSM_soft_parameters calc_beta() const;
10 };
```

where the name `MSSMcbs` is an abbreviation of the MSSM with custom-built β's. Note that the objects for `MSSM` are reused where possible: `MSSM_input_parameters` in line 6 as well as `MSSM_soft_parameters` in line 9. This saves the programmer from excessive duplication of codes. The member function definitions read:

```
1  Eigen::ArrayXd MSSMcbs<Two_scale>::beta() const
```

```
2  {
3    return calc_beta().get();
4  }
5
6  MSSM_soft_parameters MSSMcbs<Two_scale>::calc_beta() const
7  {
8    MSSM_soft_parameters betas(MSSM<Two_scale>::calc_beta());
9    if (get_loops() <= 2) return betas;
10   double bg33 = /* formula in terms of g1, g2, g3, Yu, Yd, Ye */;
11   betas.set_g3(betas.get_g3() + Power(oneOver16PiSqr,3) * g3 * bg33);
12   return betas;
13 }
```

The full C++ expression of `bg33`, used in `MSSMcbs_two_scale_model.cpp`, has been adapted from the code by Jack and Jones [88]. This already completes the amendment of the g_3 β-function.

The next step is to write a substitute for the low-energy boundary condition class. It must be declared as a descendant of `Constraint<Two_scale>` whose function is described in Section 6.2:

```
1  #include "two_scale_constraint.hpp"
2
3  template<>
4  class MSSMcbs_low_scale_constraint<Two_scale> : public Constraint<Two_scale> {
5  public:
6    MSSMcbs_low_scale_constraint(const MSSM_input_parameters&, const QedQcd&);
7    virtual ~MSSMcbs_low_scale_constraint();
8    void set_threshold_corrections(unsigned);
9    ...
10 private:
11   MSSMcbs<Two_scale>* model;
12   QedQcd oneset;
13   double new_g3;
14   unsigned threshold_corrections;
15   void calculate_DRbar_gauge_couplings();
16   double calculate_alS5DRbar_over_alS5MSbar(double) const;
17   double calculate_zeta_g_QCD_2(double) const;
18   double calculate_zeta_g_SUSY_2(double) const;
19   ...
20 };
```

In line 11, the type of `model` has been adapted to the new model. In fact, this class should work even if `model` remained a pointer to `MSSM<Two_scale>` because of inheritance. The main additions to `MSSM_low_scale_constraint` in `models/MSSM/` are the member functions in lines 16–18, which evaluate the two-loop matching coefficients from Ref. [89]. The following member function then performs the two-step decoupling as reported in this reference:

```
1  void MSSMcbs_low_scale_constraint<Two_scale>::calculate_DRbar_gauge_couplings()
2  {
3    ...
4    double alpha_s = oneset.displayAlpha(ALPHAS);
5    double alS5DRbar_over_alS5MSbar = 1;
6    double zeta_g_QCD_2 = 1;
7    double zeta_g_SUSY_2 = 1;
```

```
  if (model->get_thresholds()) {
    alS5DRbar_over_alS5MSbar = calculate_alS5DRbar_over_alS5MSbar(alpha_s);
    alpha_s *= alS5DRbar_over_alS5MSbar;  // alS5MSbar -> alS5DRbar
    zeta_g_QCD_2 = calculate_zeta_g_QCD_2(alpha_s);
    alpha_s /= zeta_g_QCD_2;              // alS5DRbar -> alS6DRbar
    zeta_g_SUSY_2 = calculate_zeta_g_SUSY_2(alpha_s);
    alpha_s /= zeta_g_SUSY_2;             // alS6DRbar -> alS6DRbarMSSM
    ...
  }
  new_g3 = Sqrt(4*Pi * alpha_s);
  ...
}
```

Finally, one can integrate the new boundary condition class `MSSMcbs_low_scale_constraint` together with the new model `MSSMcbs` into the spectrum generator in a straightforward manner. They should supersede `MSSM_low_scale_constraint` and `MSSM`, respectively. The replacement should be carried out in those objects that depend on these classes, i.e. the initial guesser:

```
template<>
class MSSMcbs_initial_guesser<Two_scale> : public Initial_guesser<Two_scale> {
public:
  MSSMcbs_initial_guesser(MSSMcbs<Two_scale>*,
                          const MSSM_input_parameters&,
                          const QedQcd&,
                          const MSSMcbs_low_scale_constraint<Two_scale>&,
                          const MSSM_susy_scale_constraint<Two_scale>&,
                          const MSSM_high_scale_constraint<Two_scale>&);
  ...
private:
  MSSMcbs<Two_scale>* model;
  MSSM_input_parameters input_pars;
  QedQcd oneset;
  MSSMcbs_low_scale_constraint<Two_scale> low_constraint;
  MSSM_susy_scale_constraint<Two_scale> susy_constraint;
  MSSM_high_scale_constraint<Two_scale> high_constraint;
  ...
};
```

as well as the spectrum generator object:

```
template <class T>
void MSSMcbs_spectrum_generator<T>::run(const QedQcd& oneset,
                                        const MSSM_input_parameters& input)
{
  ...
  MSSMcbs_initial_guesser<T> initial_guesser
    (&model, input, oneset,
     low_scale_constraint, susy_scale_constraint, high_scale_constraint);
  ...
}
```

One can find a working realization of this example in `examples/customized-betas/`.

The procedure described above is essentially all that is needed to employ new spectrum generator

components, which may be composed from scratch or through a linkage to external routines. In doing so, notice that there was an evident limit on the scope of modules which one had to deal with. For instance, it was clear from the outset that one does not have to go through the code of the central fixed-point iteration engine, `RGFlow`. This manifests the power of the clear separation among objects each with its well-defined distinct role. This is just like the fact that one does not need to access the internals of the `std::sort` function in the C++ Standard Library. It might be entertaining to complete the analogy by mapping the model objects in `RGFlow` to the elements that `std::sort` sorts and the boundary condition objects to the comparator function.

8. Tests and comparisons with other spectrum generators

8.1. Run-time comparison

One of FlexibleSUSY's design goals is a short run-time. In this section we demonstrate that this goal was achieved by comparing the run-time of two different sets of CMSSM spectrum generators:

- *Without flavour violation:* Disallowing flavour violation simplifies the calculation of the pole masses, because flavour-off-diagonal sfermion self-energy matrix elements don't need to be calculated. Here we compare FlexibleSUSY's non-flavour violating CMSSM spectrum generator FlexibleSUSY-NoFV (version 1.0.0) against SPheno (version 3.2.4) and Softsusy (version 3.4.0).

- *With flavour violation:* Allowing for flavour violation in general increases the run-time of spectrum generators, because the full 6×6 sfermion self-energy matrices have to be calculated. Here we compare FlexibleSUSY-FV (version 1.0.0) and SPhenoMSSM (generated with SARAH 4.1.0 and linked against SPheno 3.2.4). Both spectrum generators are based on SARAH's MSSM model file, which allows for flavour violation.

FlexibleSUSY and Softsusy are compiled with g++ 4.8.0 and Intel ifort 13.1.3 20130607. SPheno and SPhenoMSSM are compiled with Intel ifort 13.1.3 20130607.[3]

For the run-time comparison we're generating $2 \cdot 10^4$ random CMSSM parameter points with $m_0 \in [50, 1000]\,\text{GeV}$, $m_{1/2} \in [50, 1000]\,\text{GeV}$, $\tan\beta \in [1, 100]$, $\text{sign}\,\mu \in \{-1, +1\}$ and $A_0 \in [-1000, 1000]\,\text{GeV}$. For each point an SLHA input file is created by appending the values of m_0, $m_{1/2}$, $\tan\beta$, $\text{sign}\,\mu$, A_0 in form of a `MINPAR` block to the SLHA template file given in Appendix A. The resulting SLHA input file is passed to each spectrum generator and the (wall-clock) time is measured until the program has finished. The average run-times for three different CPU types can be found in Table 1. The first column shows the run-time on a Intel Core2 Duo (P8600, 2.40 GHz) where only one core was enabled. The second column shows the run-time on the same processor where both cores were enabled. In the third column a machine with two Intel Xeon CPUs (L5640, 2.27 GHz, 6 cores) was used.

Under both the non-flavour violating spectrum generators (first three rows) as well as the flavour violating ones (4th and 5th row) we find that FlexibleSUSY is significantly fastest. Compared to SPheno, FlexibleSUSY-NoFV is faster by a factor 1.4–1.7, and compared to Softsusy around a factor 2–2.5. Under the flavour violating spectrum generators FlexibleSUSY-FV is faster

[3]Intel's ifort compiler decreases the run-time of SPheno and SPhenoMSSM by approximately a factor 1.5, compared to gfortran.

	Intel Core2 Duo (P8600, 1 core)	Intel Core2 Duo (P8600, 2 cores)	2 × Intel Xeon (L5640, 6 cores)
FlexibleSUSY-NoFV 1.0.0	0.086 s	0.079 s	0.060 s
SPheno 3.2.4	0.119 s	0.114 s	0.101 s
Softsusy 3.4.0	0.175 s	0.171 s	0.147 s
FlexibleSUSY-FV 1.0.0	0.150 s	0.113 s	0.074 s
SPhenoMSSM 4.1.0	0.415 s	0.401 s	0.370 s

Table 1: Average run-time of CMSSM spectrum generators for random parameter points. The first three rows show spectrum generators which disallow flavour violation. Rows 4–5 contain flavour violating spectrum generators, based on SARAH's MSSM model file.

than SPhenoMSSM by a factor 2.8–5. Reason for the long run-time of SPhenoMSSM is the long calculation duration of the two-loop β-functions. Here FlexibleSUSY benefits a lot from Eigen's well-optimizable matrix expressions. We also find that increasing the number of CPU cores reduces the run-time of FlexibleSUSY. The reason is that FlexibleSUSY calculates each pole mass in a separate thread, and therefore benefits from multi-core CPUs.

8.2. Numeric tests

To check the correctness of FlexibleSUSY's generated spectrum generators extensive unit testing against Softsusy's MSSM and NMSSM implementations (both Z_3-invariant and Z_3-violating variants) was carried out. We checked mass matrices, EWSB equations, one- and two-loop β-functions, one- and two-loop self-energies and one- and two-loop tadpoles and found all to agree within double machine precision. We also compared the overall pole mass spectrum after the full fixed-point iteration has finished, and found the spectra to agree at the sub-permille level.

FlexibleSUSY generated NUHM E_6SSM has also been compared against a handwritten spectrum generator for a constrained version of the E_6SSM [90, 91, 92, 93, 94, 95]. The RGEs could be compared directly in unit tests and were found to match. The handwritten code doesn't include full one-loop self-energies or tadpoles so tests on these were not carried out. Although the generators assume different constraints they could be compared by using the output of the CE_6SSM generator as an input to FlexibleSUSY and the spectra were found to be in reasonable agreement, given the different levels of precision.

In addition FlexibleSUSY has already undergone some user testing. This includes analytic tests of the R-symmetric low-energy model (MRSSM) and alternative E_6-inspired SUSY scenarios. The users who have helped us with this are thanked in the acknowledgements.

We also compared the run-time of FlexibleSUSY against SPheno, Softsusy and the SARAH generated MSSM spectrum generator SPhenoMSSM. The test results can be found in Section 8.1.

9. Conclusions

We have presented FlexibleSUSY, a Mathematica and C++ package, which generates fast and modular spectrum generators for any user specified SUSY model. FlexibleSUSY is distributed with a large number of predefined models for the CMSSM, NMSSM, USSM, E_6SSM, MRSSM etc., which can be generated immediately without any editing. In particular the CMSSM and NMSSM

spectrum generators constitute a fast and reliable alternative to the existing publicly available spectrum generators, Softsusy, SPheno and NMSPEC.

We have described how the generated source code can be influenced at two different levels: The Mathematica level where the user provides a model file, and the C++ level where the generated objects can be easily exchanged, extended, modified and reused. This provides great flexibility for creating custom spectrum generators for both the most common and most extraordinary models. We have demonstrated these features in detailed examples for the NUHMSSM, right handed neutrinos and on adding three-loop RGEs and two-loop matching for the strong gauge coupling.

The generated code has been extensively tested against Softsusy, and additional tests have been carried out for non-minimal models, the E_6SSM and MRSSM. Speed tests have also been performed against Softsusy, SPheno and SPheno-like MSSM code generated by SARAH, demonstrating that FlexibleSUSY runs faster than all three.

As a result FlexibleSUSY enables fast exploitation of new SUSY models with high precision and reliability.

Acknowledgments

A.V. would like to thank Florian Staub for countless explanations of SARAH's internals, discussions, and exceptionally fast bug fixing. P.A. would like to thank Roman Nevzorov for useful discussions about challenges in non-minimal SUSY models. The authors would also like to thank Lewis Tunstall for helping with early tests against Next-to-Minimal Softsusy; Sophie Underwood for discovering problems when introducing couplings with a single family index; Philip Diessner for testing and identifying several bugs in FlexibleSUSY and in SARAH, in work on the MRSSM; Ulrik Günther for compilation tests on Mac OS X and Dylan Harries for spotting a bug in the configuration script. J.P. acknowledges support from the MEC and FEDER (EC) Grants FPA2011–23596 and the Generalitat Valenciana under grant PROMETEOII/2013/017. This work has been supported by the German Research Foundation DFG through Grant No. STO876/2-1.

Appendix A. Speed test SLHA input file

```
Block MODSEL                # Select model
    6    0                  # flavour violation
    1    1                  # mSUGRA
Block SMINPUTS              # Standard Model inputs
    1   1.279180000e+02     # alpha^(-1) SM MSbar(MZ)
    2   1.166390000e-05     # G_Fermi
    3   1.189000000e-01     # alpha_s(MZ) SM MSbar
    4   9.118760000e+01     # MZ(pole)
    5   4.200000000e+00     # mb(mb) SM MSbar
    6   1.709000000e+02     # mtop(pole)
    7   1.777000000e+00     # mtau(pole)
Block SOFTSUSY              # SOFTSUSY specific inputs
    1   1.000000000e-04     # tolerance
    2   2                   # up-quark mixing (=1) or down (=2)
    3   0                   # printout
    5   1                   # 2-loop running
    7   2                   # EWSB and Higgs mass loop order
Block FlexibleSUSY
```

```
    0   1.000000000e-04      # precision goal
    1   0                    # max. iterations (0 = automatic)
    2   0                    # algorithm (0 = two_scale, 1 = lattice)
    3   0                    # calculate SM pole masses
    4   2                    # pole mass loop order
    5   2                    # EWSB loop order
    6   2                    # beta-functions loop order
Block SPhenoInput            # SPheno specific input
    1  -1                    # error level
    2   1                    # SPA conventions
    11  0                    # calculate branching ratios
    13  0                    # include 3-Body decays
    12  1.000E-04            # write only branching ratios larger than this value
    31  -1                   # fixed GUT scale (-1: dynamical GUT scale)
    32  0                    # Strict unification
    34  1.000E-04            # Precision of mass calculation
    35  40                   # Maximal number of iterations
    37  1                    # Set Yukawa scheme
    38  2                    # 1- or 2-Loop RGEs
    50  1                    # Majorana phases: use only positive masses
    51  0                    # Write Output in CKM basis
    52  0                    # Write spectrum in case of tachyonic states
    55  1                    # Calculate one loop masses
    57  0                    # Calculate low energy constraints
    60  0                    # Include possible, kinetic mixing
    65  1                    # Solution tadpole equation
    75  0                    # Write WHIZARD files
    76  0                    # Write HiggsBounds file
    86  0.                   # Maximal width to be counted as invisible in Higgs \
  decays
    510 0.                   # Write tree level values for tadpole solutions
    515 0                    # Write parameter values at GUT scale
    520 0.                   # Write effective Higgs couplings (HiggsBounds \
  blocks)
    525 0.                   # Write loop contributions to diphoton decay of Higgs
Block MINPAR
    1   [50..1000]           # m0(MX)
    2   [50..1000]           # m12(MX)
    3   [1..100]             # tan(beta)(MZ) DRbar
    4   {-1,+1}              # sign(mu)
    5   [-1000..1000]        # A0(MX)
```

[1] S. R. Coleman and J. Mandula, Phys. Rev. **159**, 1251 (1967).
[2] R. Haag, J. T. Lopuszanski and M. Sohnius, Nucl. Phys. B **88**, 257 (1975).
[3] S. Weinberg, Phys. Rev. D **13**, 974 (1976).
[4] S. Weinberg, Phys. Rev. D **19** (1979) 1277.
[5] E. Gildener, Phys. Rev. D **14**, 1667 (1976).
[6] L. Susskind, Phys. Rev. D **20** (1979) 2619.
[7] G. 't Hooft, C. Itzykson, A. Jaffe, H. Lehmann, P. K. Mitter, I. M. Singer and R. Stora, NATO Adv. Study Inst. Ser. B Phys. **59**, pp.1 (1980).
[8] P. Langacker, In *Boston 1990, Proceedings, Particles, strings and cosmology* 237-269 and Pennsylvania Univ. Philadelphia - UPR-0435T (90,rec.Oct.) 33 p. (015721) (see HIGH ENERGY PHYSICS INDEX 29 (1991) No. 9950)
[9] J. R. Ellis, S. Kelley and D. V. Nanopoulos, Phys. Lett. B **260**, 131 (1991).
[10] U. Amaldi, W. de Boer and H. Furstenau, Phys. Lett. B **260**, 447 (1991).
[11] P. Langacker and M. -x. Luo, Phys. Rev. D **44**, 817 (1991).
[12] C. Giunti, C. W. Kim and U. W. Lee, Mod. Phys. Lett. A **6**, 1745 (1991).
[13] H. Goldberg, Phys. Rev. Lett. **50**, 1419 (1983) [Erratum-ibid. **103**, 099905 (2009)].
[14] J. R. Ellis, J. S. Hagelin, D. V. Nanopoulos, K. A. Olive and M. Srednicki, Nucl. Phys. B **238**, 453 (1984).
[15] L. Girardello and M. T. Grisaru, Nucl. Phys. B **194** (1982) 65.
[16] B. C. Allanach, Comput. Phys. Commun. **143**, 305 (2002) [hep-ph/0104145].
[17] W. Porod, Comput. Phys. Commun. **153**, 275 (2003) [hep-ph/0301101].
[18] A. Djouadi, J. -L. Kneur and G. Moultaka, Comput. Phys. Commun. **176**, 426 (2007) [hep-ph/0211331].
[19] H. Baer, F. E. Paige, S. D. Protopopescu and X. Tata, hep-ph/9305342.
[20] D. Chowdhury, R. Garani and S. K. Vempati, Comput. Phys. Commun. **184** (2013) 899 [arXiv:1109.3551 [hep-ph]].
[21] U. Ellwanger and C. Hugonie, Comput. Phys. Commun. **177**, 399 (2007) [hep-ph/0612134].
[22] B. C. Allanach, P. Athron, L. C. Tunstall, A. Voigt and A. G. Williams, arXiv:1311.7659 [hep-ph].
[23] P. Fayet, Nucl. Phys. B **90** (1975) 104; Phys. Lett. B **64** (1976) 159; Phys. Lett. B **69** (1977) 489 and Phys. Lett. B **84** (1979) 416; H.P. Nilles, M. Srednicki and D. Wyler, Phys. Lett. B **120** (1983) 346; J.M. Frere, D.R. Jones and S. Raby, Nucl. Phys. B **222** (1983) 11; J.P. Derendinger and C.A. Savoy, Nucl. Phys. B **237** (1984) 307; A.I. Veselov, M.I. Vysotsky and K.A. Ter-Martirosian, Sov. Phys. JETP **63** (1986) 489; J.R. Ellis, J.F. Gunion, H.E. Haber, L. Roszkowski and F. Zwirner, Phys. Rev. D **39** (1989) 844; M. Drees, Int. J. Mod. Phys. A **4** (1989) 3635; U. Ellwanger, M. Rausch de Traubenberg and C.A. Savoy, Phys. Lett. B **315** (1993) 331, Z. Phys. C **67** (1995) 665 and Nucl. Phys. B **492** (1997) 307; U. Ellwanger, Phys. Lett. B **303** (1993) 271; P. Pandita, Z. Phys. C **59** (1993) 575; T. Elliott, S.F. King and P.L. White, Phys. Rev. D **49** (1994) 2435; S.F. King and P.L. White, Phys. Rev. D **52** (1995) 4183; F. Franke and H. Fraas, Int. J. Mod. Phys. A **12** (1997) 479. D. J. Miller, R. Nevzorov and P. M. Zerwas, Nucl. Phys. B **681**, 3 (2004) [hep-ph/0304049].
[24] U. Ellwanger, C. Hugonie and A. M. Teixeira, Phys. Rept. **496**, 1 (2010) [arXiv:0910.1785 [hep-ph]].
[25] M. Maniatis, Int. J. Mod. Phys. A **25**, 3505 (2010) [arXiv:0906.0777 [hep-ph]].
[26] J. E. Kim and H. P. Nilles, Phys. Lett. B **138**, 150 (1984).
[27] S. F. King, A. Merle, S. Morisi, Y. Shimizu and M. Tanimoto, arXiv:1402.4271 [hep-ph].
[28] S. F. King, R. Luo, D. J. Miller and R. Nevzorov, JHEP **0812**, 042 (2008) [arXiv:0806.0330 [hep-ph]].
[29] G. Aad *et al.* [ATLAS Collaboration], JHEP **1310**, 130 (2013) [arXiv:1308.1841 [hep-ex]].
[30] S. Chatrchyan *et al.* [CMS Collaboration], arXiv:1402.4770 [hep-ex].
[31] G. Aad *et al.* [ATLAS Collaboration], Phys. Lett. B **710**, 49 (2012) [arXiv:1202.1408 [hep-ex]].
[32] S. Chatrchyan *et al.* [CMS Collaboration], Phys. Lett. B **710**, 26 (2012) [arXiv:1202.1488 [hep-ex]].
[33] S. F. King, S. Moretti and R. Nevzorov, Phys. Rev. D **73**, 035009 (2006) [hep-ph/0510419].
[34] S. F. King, S. Moretti and R. Nevzorov, Phys. Lett. B **634**, 278 (2006) [hep-ph/0511256].
[35] S. F. King, S. Moretti and R. Nevzorov, Phys. Lett. B **650**, 57 (2007) [hep-ph/0701064].
[36] P. Athron, J. P. Hall, R. Howl, S. F. King, D. J. Miller, S. Moretti and R. Nevzorov, Nucl. Phys. Proc. Suppl. **200-202**, 120 (2010).
[37] P. Fayet, Phys. Lett. B **69** (1977) 489.
[38] D. Suematsu and Y. Yamagishi, Int. J. Mod. Phys. A **10**, 4521 (1995) [hep-ph/9411239].
[39] M. Cvetic and P. Langacker, Phys. Rev. D **54**, 3570 (1996) [hep-ph/9511378].
[40] B. de Carlos and J. R. Espinosa, Phys. Lett. B **407**, 12 (1997) [hep-ph/9705315].
[41] M. Cvetic, D. A. Demir, J. R. Espinosa, L. L. Everett and P. Langacker, Phys. Rev. D **56** (1997) 2861 [Erratum-ibid. D **58** (1998) 119905] [hep-ph/9703317].

[42] D. A. Demir and N. K. Pak, Phys. Rev. D **57**, 6609 (1998) [hep-ph/9809357].
[43] P. Langacker and J. Wang, Phys. Rev. D **58**, 115010 (1998) [hep-ph/9804428].
[44] J. Erler, P. Langacker and T. -j. Li, Phys. Rev. D **66** (2002) 015002 [hep-ph/0205001].
[45] S. Y. Choi, H. E. Haber, J. Kalinowski and P. M. Zerwas, Nucl. Phys. B **778**, 85 (2007) [hep-ph/0612218].
[46] S. W. Ham and S. K. OH, Phys. Rev. D **76**, 095018 (2007) [arXiv:0708.1785 [hep-ph]].
[47] P. Langacker, Rev. Mod. Phys. **81**, 1199 (2009) [arXiv:0801.1345 [hep-ph]].
[48] S. W. Ham, T. Hur, P. Ko and S. K. Oh, J. Phys. G **35**, 095007 (2008) [arXiv:0801.2361 [hep-ph]].
[49] J. Kalinowski, S. F. King and J. P. Roberts, JHEP **0901**, 066 (2009) [arXiv:0811.2204 [hep-ph]].
[50] A. Bharucha, A. Goudelis and M. McGarrie, arXiv:1310.4500 [hep-ph].
[51] P. Batra, A. Delgado, D. E. Kaplan and T. M. P. Tait, JHEP **0402**, 043 (2004) [hep-ph/0309149].
[52] A. D. Medina, N. R. Shah and C. E. M. Wagner, Phys. Rev. D **80**, 015001 (2009) [arXiv:0904.1625 [hep-ph]].
[53] F. Staub, W. Porod and B. Herrmann, JHEP **1010**, 040 (2010) [arXiv:1007.4049 [hep-ph]].
[54] F. Staub, Comput. Phys. Commun. **181**, 1077 (2010) [arXiv:0909.2863 [hep-ph]].
[55] F. Staub, Comput. Phys. Commun. **182**, 808 (2011) [arXiv:1002.0840 [hep-ph]].
[56] F. Staub, Computer Physics Communications **184**, pp. 1792 (2013) [Comput. Phys. Commun. **184**, 1792 (2013)] [arXiv:1207.0906 [hep-ph]].
[57] F. Staub, arXiv:1309.7223 [hep-ph].
[58] G. Degrassi, P. Slavich and F. Zwirner, Nucl. Phys. B **611** (2001) 403 [hep-ph/0105096].
[59] A. Brignole, G. Degrassi, P. Slavich and F. Zwirner, Nucl. Phys. B **631** (2002) 195 [hep-ph/0112177].
[60] A. Dedes and P. Slavich, Nucl. Phys. B **657** (2003) 333 [hep-ph/0212132].
[61] A. Brignole, G. Degrassi, P. Slavich and F. Zwirner, Nucl. Phys. B **643** (2002) 79 [hep-ph/0206101].
[62] A. Dedes, G. Degrassi and P. Slavich, Nucl. Phys. B **672** (2003) 144 [hep-ph/0305127].
[63] G. Degrassi and P. Slavich, Nucl. Phys. B **825**, 119 (2010) [arXiv:0907.4682 [hep-ph]].
[64] Eigen library, version 3.1 `http://eigen.tuxfamily.org`.
[65] P. Z. Skands, B. C. Allanach, H. Baer, C. Balazs, G. Belanger, F. Boudjema, A. Djouadi and R. Godbole *et al.*, JHEP **0407** (2004) 036 [hep-ph/0311123].
[66] B. C. Allanach, C. Balazs, G. Belanger, M. Bernhardt, F. Boudjema, D. Choudhury, K. Desch and U. Ellwanger *et al.*, Comput. Phys. Commun. **180**, 8 (2009) [arXiv:0801.0045 [hep-ph]].
[67] P. Athron, S. F. King, D. J. Miller, S. Moretti and R. Nevzorov, J. Phys. Conf. Ser. **110** (2008) 072001 [arXiv:0708.3248 [hep-ph]].
[68] G. D. Kribs, E. Poppitz and N. Weiner, Phys. Rev. D **78** (2008) 055010 [arXiv:0712.2039 [hep-ph]].
[69] F. Staub, arXiv:0806.0538 [hep-ph].
[70] D. R. T. Jones, Nucl. Phys. B **87** (1975) 127.
[71] D. R. T. Jones and L. Mezincescu, Phys. Lett. B **136** (1984) 242.
[72] P. C. West, Phys. Lett. B **137** (1984) 371.
[73] S. P. Martin and M. T. Vaughn, Phys. Lett. B **318** (1993) 331 [hep-ph/9308222].
[74] Y. Yamada, Phys. Rev. Lett. **72** (1994) 25 [hep-ph/9308304].
[75] S. P. Martin and M. T. Vaughn, Phys. Rev. D **50**, 2282 (1994) [Erratum-ibid. D **78**, 039903 (2008)] [hep-ph/9311340].
[76] R. M. Fonseca, M. Malinsky, W. Porod and F. Staub, Nucl. Phys. B **854** (2012) 28 [arXiv:1107.2670 [hep-ph]].
[77] Y. Yamada, Phys. Rev. D **50**, 3537 (1994) [hep-ph/9401241].
[78] M. Sperling, D. Stöckinger and A. Voigt, JHEP **1307** (2013) 132 [arXiv:1305.1548 [hep-ph]].
[79] M. Sperling, D. Stöckinger and A. Voigt, JHEP **1401** (2014) 068 [arXiv:1310.7629 [hep-ph]].
[80] J. Beringer *et al.* [Particle Data Group Collaboration], Phys. Rev. D **86** (2012) 010001.
[81] L. J. Hall, Nucl. Phys. B **178** (1981) 75.
[82] A. Bednyakov, A. Onishchenko, V. Velizhanin and O. Veretin, Eur. Phys. J. C **29** (2003) 87 [hep-ph/0210258].
[83] H. Baer, J. Ferrandis, K. Melnikov and X. Tata, Phys. Rev. D **66** (2002) 074007 [hep-ph/0207126].
[84] V. D. Barger, M. S. Berger and P. Ohmann, Phys. Rev. D **49**, 4908 (1994) [hep-ph/9311269].
[85] V. Berezinsky, A. Bottino, J. R. Ellis, N. Fornengo, G. Mignola and S. Scopel, Astropart. Phys. **5** (1996) 1 [hep-ph/9508249]; P. Nath and R. L. Arnowitt, Phys. Rev. D **56** (1997) 2820 [hep-ph/9701301]; A. Bottino, F. Donato, N. Fornengo and S. Scopel, Phys. Rev. D **63** (2001) 125003 [hep-ph/0010203]; V. Bertin, E. Nezri and J. Orloff, JHEP **0302** (2003) 046 [hep-ph/0210034]; M. Drees, M. M. Nojiri, D. P. Roy and Y. Yamada, Phys. Rev. D **56** (1997) 276 [Erratum-ibid. D **64** (2001) 039901] [hep-ph/9701219]; M. Drees, Y. G. Kim, M. M. Nojiri, D. Toya, K. Hasuko and T. Kobayashi, Phys. Rev. D **63** (2001) 035008 [hep-ph/0007202]; J. R. Ellis, T. Falk, G. Ganis, K. A. Olive and M. Schmitt, Phys. Rev. D **58** (1998) 095002 [hep-ph/9801445]; J. R. Ellis, T. Falk, G. Ganis and K. A. Olive, Phys. Rev. D **62** (2000) 075010 [hep-ph/0004169]; J. R. Ellis,

K. A. Olive and Y. Santoso, Phys. Lett. B **539** (2002) 107 [hep-ph/0204192]; J. R. Ellis, T. Falk, K. A. Olive and Y. Santoso, Nucl. Phys. B **652** (2003) 259 [hep-ph/0210205].
[86] P. Minkowski, Phys. Lett. B **67** (1977) 421; T. Yanagida, Proc. of the Workshop on Unified Theories and the Baryon Number of the Universe, edited by O. Sawada and A. Sugamoto, KEK, Japan (1979) 95; M. Gell-Mann, P. Ramond and R. Slansky, Supergravity, edited by F. Nieuwenhuizen and D. Friedman, North Holland, Amsterdam (1979) 315 [arXiv:1306.4669 [hep-th]]; R. N. Mohapatra and G. Senjanovic, Phys. Rev. Lett. **44** (1980) 912.
[87] F. Borzumati and A. Masiero, Phys. Rev. Lett. **57** (1986) 961.
[88] I. Jack and D. R. T. Jones, `http://www.liv.ac.uk/~dij/betas`.
[89] R. Harlander, L. Mihaila and M. Steinhauser, Phys. Rev. D **72** (2005) 095009 [hep-ph/0509048].
[90] P. Athron, S. F. King, D. J. Miller, S. Moretti and R. Nevzorov, Phys. Lett. B **681**, 448 (2009) [arXiv:0901.1192 [hep-ph]].
[91] P. Athron, S. F. King, D. J. Miller, S. Moretti and R. Nevzorov, Phys. Rev. D **80** (2009) 035009 [arXiv:0904.2169 [hep-ph]].
[92] P. Athron, S. F. King, D. J. Miller, S. Moretti and R. Nevzorov, Phys. Rev. D **84**, 055006 (2011) [arXiv:1102.4363 [hep-ph]].
[93] P. Athron, S. F. King, D. J. Miller, S. Moretti and R. Nevzorov, Phys. Rev. D **86**, 095003 (2012) [arXiv:1206.5028 [hep-ph]].
[94] P. Athron, D. Stockinger and A. Voigt, Phys. Rev. D **86** (2012) 095012 [arXiv:1209.1470 [hep-ph]].
[95] P. Athron, M. Binjonaid and S. F. King, Phys. Rev. D **87**, no. 11, 115023 (2013) [arXiv:1302.5291 [hep-ph]].