THE EUROPEAN
PHYSICAL JOURNAL C

CrossMark

SpecBit, DecayBit and PrecisionBit: GAMBIT modules for computing mass spectra, particle decay rates and precision observables

The GAMBIT Models Workgroup: Peter Athron[1,2,a], **Csaba Balázs**[1,2], **Lars A. Dal**[3], **Joakim Edsjö**[4,5],
Ben Farmer[4,5,b], **Tomás E. Gonzalo**[3], **Anders Kvellestad**[6], **James McKay**[7], **Antje Putze**[8], **Chris Rogan**[9],
Pat Scott[7,c], **Christoph Weniger**[10], **Martin White**[2,11]

[1] School of Physics and Astronomy, Monash University, Melbourne, VIC 3800, Australia
[2] Australian Research Council Centre of Excellence for Particle Physics at the Tera-scale, Australia, http://www.coepp.org.au/
[3] Department of Physics, University of Oslo, 0316 Oslo, Norway
[4] Oskar Klein Centre for Cosmoparticle Physics, AlbaNova University Centre, 10691 Stockholm, Sweden
[5] Department of Physics, Stockholm University, 10691 Stockholm, Sweden
[6] NORDITA, Roslagstullsbacken 23, 10691 Stockholm, Sweden
[7] Department of Physics, Blackett Laboratory, Imperial College London, Prince Consort Road, London SW7 2AZ, UK
[8] LAPTh, Université de Savoie, CNRS, 9 chemin de Bellevue, B.P.110, 74941 Annecy-le-Vieux, France
[9] Department of Physics, Harvard University, Cambridge, MA 02138, USA
[10] GRAPPA, Institute of Physics, University of Amsterdam, Science Park 904, 1098 XH Amsterdam, The Netherlands
[11] Department of Physics, University of Adelaide, Adelaide, SA 5005, Australia

Abstract We present the GAMBIT modules SpecBit, DecayBit and PrecisionBit. Together they provide a new framework for linking publicly available spectrum generators, decay codes and other precision observable calculations in a physically and statistically consistent manner. This allows users to automatically run various combinations of existing codes as if they are a single package. The modular design allows software packages fulfilling the same role to be exchanged freely at runtime, with the results presented in a common format that can easily be passed to downstream dark matter, collider and flavour codes. These modules constitute an essential part of the broader GAMBIT framework, a major new software package for performing global fits. In this paper we present the observable calculations, data, and likelihood functions implemented in the three modules, as well as the conventions and assumptions used in interfacing them with external codes. We also present 3-BIT-HIT, a command-line utility for computing mass spectra, couplings, decays and precision observables in the MSSM, which shows how the three modules can easily be used independently of GAMBIT.

a e-mail: peter.athron@coepp.org.au
b e-mail: benjamin.farmer@fysik.su.se
c e-mail: p.scott@imperial.ac.uk

Contents

Springer

1 Introduction

Run II of the Large Hadron Collider (LHC) is engaged in a wide-ranging search for evidence of physics Beyond the Standard Model (BSM). Such models typically have large parameter spaces, so understanding their phenomenology requires detailed calculations using computer programs. Phenomenological studies therefore often involve a large chain of public codes that must be linked together.

This set of codes includes spectrum generators (to determine the masses of new particles), decay calculators (to obtain decay widths), and packages capable of predicting low-energy precision observables, such as the anomalous magnetic moment of the muon. These codes need to be linked together in such a way that information from the spectrum generator can be passed to the other calculators, and their outputs can in turn be used in other programs.

For the Minimal Supersymmetric Standard Model (MSSM) and the Next-to-minimal Supersymmetric Standard Model (NMSSM), there are the SLHA [1] and SLHA2 [2] conventions, which simplify matters somewhat. However, even in these cases it can be far more convenient to have the programs automatically linked, as can be testified by the popularity of packages that incorporate individual codes, such as SUSY-HIT [3] for the MSSM and NMSSMTools [4–6] for the NMSSM. On the other hand, keeping individual codes distinct in a modular framework allows for easier isolation of the origin of differences in results, and for the ability to mix and match different codes according to preference or the specific advantages of one tool or another. Indeed, the

need to compare several software packages for cross-checks, catching bugs and revealing uncertainties in each calculation has been demonstrated on numerous occasions (see e.g. Refs. [7–10]).

Here we present a framework that provides the best of both worlds, designed to work with all models including both non-supersymmetric models and exotic SUSY models beyond the MSSM and NMSSM. The framework consists of three packages: SpecBit, for handing renormalisation group running and calculation of mass spectra, DecayBit, for computing branching ratios and widths, and PrecisionBit, for calculating other precision observables. All three packages are interfaced in a common way within GAMBIT 1.0.0 [11], allowing specific functions and external software to be exchanged at runtime, whilst being run from exactly the same input parameters under the same physical conventions. The user interface is designed to be as simple and general as possible, so that information can be extracted and inserted in an intuitive way.

GAMBIT (the Global and Modular Beyond-SM Inference Tool) is a multi-purpose physics tool for performing parameter scans and global fits in BSM models using either frequentist or Bayesian statistics. Each of the three packages we describe here is a **module** within the broader GAMBIT framework. Here we **highlight** terms that may be considered GAMBIT jargon, and provide a simple summary of their meanings in the **glossary** (Appendix H). Each module collects a series of **module functions** based on a common theme. Each module function typically calculates a single observable or likelihood component for use in a fit, or performs some calculation that is needed by another module function in order to eventually arrive at the value of an observable or likelihood. Module functions that require the results of other module functions can declare **dependencies** on the physical or mathematical quantities that they require. GAMBIT defines a series of theoretical models available for analysis, and various rules that relate the different module functions to each other, to models, and to functions that are available for modules to call from external physics codes, known as **backends**. At runtime, the GAMBIT **dependency resolver** identifies the module and backend functions required to compute the observables and likelihoods requested by a user, and arranges them into a **dependency tree**. It then uses methods from graph theory to 'solve' the tree, and determine the order in which the functions must be called so that all dependencies and model consistency requirements are guaranteed to be satisfied. GAMBIT's sampling module ScannerBit [12] runs the user's choice of sampling algorithm on the solved dependency tree, and saves the resulting parameter samples, derived observables and likelihoods in an output database for subsequent statistical analysis and plotting.

SpecBit, DecayBit and PrecisionBit are used to compute and return spectra, decay widths and precision observables, run associated **backend** codes, interpret the results

thus obtained, and provide them to other **modules** as required for subsequent calculations. Other GAMBIT modules calculate dark matter observables (DarkBit [13]), high-energy collider signatures (ColliderBit [14]) and quantities from flavour physics (FlavBit [15]). An extended description of the structure, features and abilities of GAMBIT can be found in Ref. [11].

SpecBit, DecayBit and PrecisionBit are designed so that a user may easily add new models and interfaces to new backends. Models that have already been implemented in GAMBIT are automatically supported. The first release features various incarnations of the MSSM and the scalar-singlet dark matter model, as these are the models on which the first GAMBIT scans [16–18] have been performed.

The three modules have been initially set up to exploit several external codes:[1]

- FlexibleSUSY [19] and SPheno [20,21] for spectrum generation, performing calculations in the dimensional reduction ($\overline{DR}$) scheme.
- FeynHiggs [22–27] for additional Higgs and W mass calculations in a mixed $\overline{DR}$/on-shell (OS) scheme.
- HDECAY[2] [28–30] and FeynHiggs [22–26] for Higgs decays.
- SDECAY[2] [31] for sparticle decays
- FeynHiggs, SuperIso [32–34] and GM2Calc [35] for calculation of additional precision observables.

The models and backends supported by SpecBit, DecayBit and PrecisionBit will be continually updated, and they are expected to grow rapidly. There are immediate plans to add interfaces to the spectrum generators SOFTSUSY [36–39] and SuSpect [40] for the MSSM, next-to-minimal SOFTSUSY [41], NMSSMCALC [42–45] and NMSPEC [6] for the NMSSM, and many additional models via FlexibleSUSY [19] and SARAH / SPheno [21,46–50]. For decays in the MSSM, HFOLD [51] and SFOLD [52] could also be added. We are also adding SUSYHD [53] as a first example for a forthcoming Mathematica[3] interface and we will also add similar pure and hybrid effective field theory calculations of HSSUSY [54,55] and FlexibleEFTHiggs [55] from FlexibleSUSY. NMSSM decays will be obtainable from NMSSMCALC [42–45] and NMHDECAY [4,5]. Decays in other models can be added from SARAH /

---

[1] When any of these external codes are used as part of GAMBIT or the SpecBit, DecayBit and PrecisionBit modules, the references for that code listed here should be cited along with this manual. A full list of references associated with all external codes currently utilised as GAMBIT backends can be found in the file README.md in the main directory of the GAMBIT source code.

[2] We interface with SDECAY and HDECAY via SUSY-HIT [3].

[3] http://www.wolfram.com/mathematica.

SPheno and the upcoming FlexibleDecay extension of FlexibleSUSY.

In Sect. 2 we describe the module SpecBit. This includes how SpecBit manages the running of spectrum generators, how to access and use the information it extracts from them, and how it can be used to analyse vacuum stability. Next we provide a detailed description of the module DecayBit (Sect. 3), including the decay data it contains, decay calculations it performs internally, and its use of backends. In Sect. 4 we detail PrecisionBit, including its likelihood functions and interfaces to external precision codes. Section 5 gives some examples of each of the modules in action within GAMBIT, and presents 3-BIT-HIT, a simple utility that performs a similar function to SUSY-HIT, and serves as a basic example of the standalone use of SpecBit, DecayBit and PrecisionBit outside of the GAMBIT framework. We give a brief summary in Sect. 6. In Appendix A we review the physics of the models discussed in this paper, and the conventions that we adopt for them. We provide details of the interface with SPheno in Appendix B, explicit documentation of some of the classes involved in SpecBit in Appendices C–E, instructions for adding new spectrum generators to SpecBit in Appendix F, and an explicit example of how to build spectrum classes for new models in Appendix G. In Appendix H, we give the glossary of GAMBIT terms highlighted at various points in this paper.

SpecBit, DecayBit and PrecisionBit are released under the standard 3-clause BSD license,[4] and can be downloaded from gambit.hepforge.org.

2 SpecBit

All information in GAMBIT about the spectrum of particle masses and their couplings comes from module functions in SpecBit. This includes the pole masses and mixings of all physical states in the model, scheme dependent parameters, such as those defined in $\overline{DR}$ [57–59] or $\overline{MS}$ [60] schemes,[5] Higgs couplings and basic SM inputs (such as the top quark

---

[4] http://opensource.org/licenses/BSD-3-Clause. Note that fjcore [56] and some outputs of FlexibleSUSY [19] (incorporating routines from SOFTSUSY [36]) are also shipped with GAMBIT 1.0. These code snippets are distributed under the GNU General Public License (GPL; http://opensource.org/licenses/GPL-3.0), with the special exception, granted to GAMBIT by the authors, that they do not require the rest of GAMBIT to inherit the GPL.

[5] Note that, for a given model, if a user wants to support backend codes that require input parameters in a different scheme to the one implemented in SpecBit, then the user must add a function that converts between the two schemes. Consistency can then be ensured by specifying a scheme-specific spectrum as a **dependency**. Currently there are no conversions between different schemes in SpecBit, because all backends and native calculations can accept $\overline{DR}$ and $\overline{MS}$ parameters as inputs.
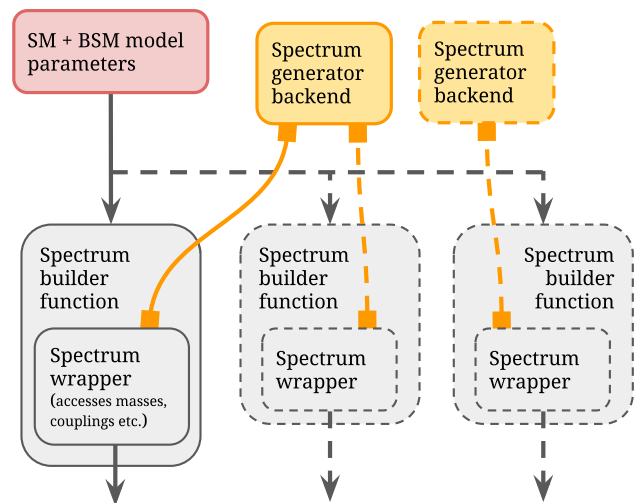
Fig. 1 Schematic representation of the structure of SpecBit. Model parameters (red box) are obtained from the GAMBIT Core and fed to SpecBit module functions (grey boxes). These module functions run spectrum generator backend codes (yellow boxes) and embed the results within `Spectrum` wrapper objects (inner grey boxes), which carry the spectrum information out to other GAMBIT functions that require it. In some cases the backend code remains connected to the wrapper object, allowing RGEs to be called to translate couplings and mass parameters to other scales. In this diagram solid outlines indicate "active" elements for a hypothetical scan, while dashed outlines indicate inactive elements. The "active" elements are those that are activated by the central GAMBIT dependency resolution system, or manually run in a standalone code. The "inactive" elements represent alternate calculation pathways not required in a scan, but can be switched on instead if the user chooses

pole mass). In BSM models where some of this required information is a calculable prediction of the model, SpecBit will obtain it by calling a spectrum generator, taking inputs as model parameters. In cases where the information is not a prediction, but is already specified directly by the free parameters of the model (e.g. if a pole mass is defined as a model parameter), SpecBit will simply take this information from the model parameters and store it in a similar way to the information extracted from the spectrum generator. Spectrum data is then stored inside `Spectrum` wrapper objects, which transport the data to other GAMBIT module functions that request it. A general overview of this process is shown in Fig. 1.

Because some calculations require the values of running parameters at a particular scale, SpecBit also allows the $\overline{DR}$ or $\overline{MS}$ parameters to be run to a scale chosen locally in any module function, using a relevant spectrum generator backend.

2.1 Supported models and spectrum generators

The first release of SpecBit is extendable to any model, but has built-in support for the scalar-singlet dark matter model, and the MSSM. It also provides a low-energy spectrum object containing SM information.

Table 1 SM **capabilities** provided by SpecBit. The **capability** `SM_spectrum` is understood to provide running parameters in the $\overline{MS}$ scheme, along with pole masses

Capability	Function (return type): brief description	Dependencies
`SMINPUTS`	`get_SMINPUTS` (`SMInputs`): Provide Standard Model parameters in SLHA2 input conventions	
`qedqcd_subspectrum`	`get_QedQcd_spectrum` (`const SubSpectrum*`): Create `QedQcdWrapper` version of `SubSpectrum` from `SMInputs` structures	`SMINPUTS`
`SM_spectrum`	`get_SM_spectrum` (`Spectrum`): Create `Spectrum` object from `QedQcdWrapper` and SM Higgs parameters	`SMINPUTS`
`Higgs_Couplings`	`SM_higgs_couplings` (`HiggsCouplingsTable`): Construct a table of SM Higgs couplings	`Higgs_decay_rates`

2.1.1 Standard model spectrum

GAMBIT is designed to fit models of BSM physics. There is thus no full spectrum generator implemented for the SM. However, BSM spectrum generators typically rely on SM inputs as low-energy boundary conditions. To store this information, SpecBit contains a `QedQcd` spectrum object, which was originally part of SOFTSUSY and is also used in FlexibleSUSY. This is the source of some of the SM data that can be extracted from SpecBit[6] and details of the relevant **capabilities** are given in Sect. 2.2.2 and Table 1. More details on this `QedQcd` object can be found in the SOFTSUSY manual [36].

2.1.2 Spectrum generators for the MSSM

MSSM mass spectra are typically obtained by finding solutions to the renormalisation group equations (RGEs) that simultaneously satisfy boundary conditions (BCs) at high and low scales, before using self energies to calculate the pole masses. The low-scale BC matches the spectrum to observed SM data, and the high-scale BC places constraints on the soft SUSY-breaking masses.

The first version of SpecBit comes with interfaces to two different spectrum generators: FlexibleSUSY and SPheno. It also contains an interface to FeynHiggs, which can be used to obtain Higgs and sparticle pole masses; details of our interface to FeynHiggs are given in Sect. 3.1.3. FlexibleSUSY creates a spectrum generator for a given model, defined in a SARAH input file, and uses a FlexibleSUSY model file to specify the BCs for that model. SPheno runs in different modes according to the input BCs.

The model is defined by both a set of input parameters that specify $\overline{DR}$ parameters of the MSSM, and an additional scale at which the soft SUSY-breaking $\overline{DR}$ parameters are defined. Note that when there are constraints relating the parameters to each other, varying this scale is not equivalent to a reparametrisation, as there will be different mass splittings that cannot be reproduced by any point in parameter space when the constraints are applied at a different scale.

SpecBit currently implements several spectrum generators using FlexibleSUSY 1.5.1 (with SARAH 4.9.1). In the spectrum generator `MSSMatMGUT`, the soft parameters are defined at the scale where the gauge couplings unify, which is determined iteratively. In `MSSMatQ`, the soft parameters are instead defined at a user-specified scale Q. The `MSSMatMGUT` and `MSSMatQ` spectrum generators support the MSSM63atMGUT and MSSM63atQ models, respectively. These models represent the most general formulation of the CP-conserving MSSM with the couplings given in Eq. (A.12f) set to zero (see Appendix A.2 for the full MSSM Lagrangian). MSSM63atMGUT is defined at the scale where the gauge couplings unify, and MSSM63atQ is defined at a user-specified scale Q. These are currently the most general SUSY models of the GAMBIT model hierarchy (which is described in Sect. 5 of Ref. [11]).

More constraining BCs create lower-dimensional subspaces of these more general parameter spaces, so a vast number of possible subspaces exist. SpecBit uses the GAMBIT model hierarchy to relate these subspaces to the MSSM63atMGUT or MSSM63atQ models, and can use the `MSSMatMGUT` or `MSSMatQ` spectrum generators to determine their mass spectra. It is also possible to directly implement the boundary condition in a FlexibleSUSY spectrum generator. As an example of this, we have also implemented a specific `CMSSM` spectrum generator. This was mostly introduced as a basic check of the model hierarchy, but remains in GAMBIT for convenience.

Technical details of how the FlexibleSUSY spectrum generators are implemented are given in documentation

---

[6] The SM data extractable from SpecBit are shown in Table 16.

shipped with the code (see `doc/Adding_FlexibleSUSY_Models.txt`), though an illustrative example is given in Appendix G.1, and details required to use them in scans are given in Sects. 2.2.4 and 2.2.5. In the additional documentation, we also demonstrate how to add an MSSM variant where soft parameters are fixed at the SUSY scale (`lowMSSM` in the FlexibleSUSY naming scheme), where the SUSY scale is defined as the geometric mean of the $\overline{DR}$ stop masses and is determined iteratively. We will include this variant natively in the next version of SpecBit.

In the case of SPheno, the running mode is determined by the initialisation of its internal variables. In contrast to FlexibleSUSY, the available modes are triggered according to the model being scanned (CMSSM, or one of the MSSMatMGUT or MSSMatQ models), not by the usage of different functions. Other running modes available in the out-of-the-box version of SPheno, such as the NMSSM, GMSB, AMSB, etc., are not covered in the backend version of the software used in GAMBIT but, as mentioned before, there are immediate plans to include these in future releases. The specific details of how to use SPheno as a spectrum generator in GAMBIT are provided in Sects. 2.2.4 and 2.2.6. A comprehensive description of the backend system can be found in the main GAMBIT paper [11], and in Appendix B for the specific case of SPheno.

For each of these spectrum generators the EWSB conditions are used to fix $|\mu|$ and b, so that the Higgs vacuum expectation values (VEVs) are fixed by the measured Z boson mass and the input ratio of the two VEVs ($\tan \beta = \frac{v_u}{v_d}$).

The pole masses and mixings calculated by the spectrum generator are stored internally in the GAMBIT `Spectrum` wrapper in a format that follows SLHA2 conventions [2] allowing full family mixing for sfermions.

2.1.3 Spectrum generators for the scalar-singlet dark matter model

The scalar-singlet mass spectrum is available in two forms. One is a via a simple container object that is set up without any spectrum generation, in other words the relevant pole masses and coupling, which govern the new physics effects at the TeV scale, are input directly from scanner-generated parameters. The other option is a spectrum object that interfaces with FlexibleSUSY to calculate the spectrum with full RGE-running capabilities.

The simple container spectrum is the most efficient option for a range of calculations in GAMBIT that require only the masses and coupling at a fixed scale. In this case, the only new model parameters required for other module functions are m_S and λ_{hs}. In the container object these parameters are stored directly from the GAMBIT model parameters.

If radiative corrections and/or renormalisation group running of model parameters is necessary, then a fully calculable

spectrum is required. For this purpose, we offer a spectrum object that uses FlexibleSUSY and the input parameters to calculate pole masses and couplings. This spectrum object also has the capacity to evolve parameters between different scales using RGEs. This spectrum object interprets the input model parameters, in particular the masses of the scalar singlet (m_S) and the Higgs boson (m_h), as $\overline{MS}$ quantities, while all other SM masses are taken according to SLHA2 conventions [2]. These running parameters are defined at the scale m_Z. The EWSB conditions are then imposed to calculate the value of the quartic Higgs coupling λ (Eq. (A.2)), and pole masses are calculated at a user-specified scale Q. The scale Q may be varied as a nuisance parameter in the GAMBIT model `SM_Higgs_running` (see Ref. [11] for more details), or left as the top mass by default. Optimally, this scale should be set to minimise logarithmic contributions to the RGE β functions with the largest coefficient.

2.2 User interface and options

2.2.1 General settings

In this section we describe how to use SpecBit to run available spectrum generators via a GAMBIT input YAML file.[7] At the most rudimentary level, spectrum information can simply be written to disk for one model point in an SLHA2-like format, using SLHAea.[8] At a more advanced level, the information can be written to disk for every point in a GAMBIT scan, via the GAMBIT printer system (for details, see Sect. 9 of the GAMBIT Core paper and manual [11]), and analysed using external software. More advanced usage, such as accessing spectrum information at the C++ level (in a GAMBIT module function, for example), is described in Sect. 2.3. For details on YAML usage of the **capabilities** and associated options described here, please see the GAMBIT manual [11] or the SpecBit example files in the `yaml_files` directory of the GAMBIT source tree. A `README` file can be found in this directory, which explains the example files further.

2.2.2 Standard model

The **capabilities** available in SpecBit related to the SM are given in Table 1. The **capability** `SM_spectrum` is provided by the function

    `get_SMINPUTS`.

---

[7] YAML is the language that GAMBIT input files are written in; see www.yaml.org.

[8] SLHAea is a C++ class allowing internal representation of SLHA files; see fthomas.github.io/slhaea.

This creates an object containing the low-energy SM parameters obtained directly from the model parameters, which must contain the parameters in the StandardModel_SLHA2 GAMBIT model [11]. This can then be used to build BSM spectrum objects that require this low-energy data.

By default, the `get_SMINPUTS` function populates its W mass with the observed value of 80.385 GeV [61]. Users who prefer the W mass returned to respect the tree-level relationship with m_Z and $\sin^2 \theta_W$ can choose for it to be calculated at tree level instead, by setting the option `enforce_tree_level_MW = true` for this function.

The module function

    `get_QedQcd_spectrum`

has a **dependency** on `SM_spectrum`, and provides the **capability** `qedqcd_subspectrum`. This function creates an effective `QedQcd` object. The returned `QedQcd` object contains the low-energy data, where all running quantities from the `SMinputs` **dependency** are now given at the scale m_Z. This is then wrapped along with the original `SMinputs` into a `SubSpectrum` object.

Finally, to provide the **capability** `SM_spectrum`, the module function

    `get_SM_spectrum`

wraps the `qedqcd_subspectrum` object into a full spectrum object along with a simple container for the Higgs pole mass and vacuum expectation value, providing a complete SM spectrum.

The **capability** `Higgs_Couplings` is described in Sect. 2.6.

2.2.3 Scalar-singlet dark matter

The **capabilities** available in SpecBit that either produce or depend on the scalar-singlet spectrum are given in Table 3.

SpecBit has a **capability** `SingletDM_spectrum` to provide essential details as regards the scalar-singlet spectrum. This **capability** is provided by the module functions

    `get_SingletDM_spectrum_simple` and
    `get_SingletDM_spectrum_FS`,

both of which return a `Spectrum` object. The former returns the simple container spectrum, as described in Appendix F.2, and the latter returns a more sophisticated spectrum, computed with FlexibleSUSY, which can run parameters as described in Sect. 2.3.3. All options described in Sect. 2.2.5 can be given to this function.

If an error is encountered during initial spectrum generation, a FlexibleSUSY error is passed to GAMBIT. The handling of such an error is controlled by the option

`invalid_point_fatal`; if this option is set to `true`, invalid spectra will trigger a SpecBit error. It is a common consideration to check for perturbativity of the dimensionless couplings up to a specific high-energy scale [62–64]. We provide the ability to do this with the option `check_perturb`. The maximum scale to run the couplings up to is set with the option `check_high_scale`. If the couplings are found to be non-perturbative, then an invalid point exception is raised as if the initial spectrum calculation encountered an error. The options, input types and default values for the `get_SingletDM_spectrum_FS` function are given in the list below.

- `check_perturb`: takes a `bool` to demand that the spectrum be run to `check_high_scale`. Default `false`.
- `check_high_scale`: takes a `double` to define the scale that the couplings are run to (given in units of GeV) after spectrum generation. Default $M_{\mathrm{Pl}} = 1.22 \times 10^{19}$ GeV.
- `invalid_point_fatal`: kill the whole scan if a FlexibleSUSY error is encountered during spectrum generation. Default `false`.
- FlexibleSUSY options: see Sect. 2.2.5.

The module function

    `get_SingletDM_spectrum_as_map`

also provides **capability** `SingletDM_spectrum`, but as a C++`map` (aliased as `map_str_dbl`). For this function to run, the `SingletDM_spectrum` needs to be provided as a `Spectrum` object. In other words, this function just translates between the `Spectrum` type and a C++ map. The main use of this function is to print the contents of the spectrum to an output stream for each data point during a scan (as the `map_str_dbl` type is "printable" but the `Spectrum` type is not), as shown in Fig. 2.

There are also some additional **capabilities** related to vacuum stability in the scalar-singlet model: `vacuum_stability`, `check_perturb_min_lambda`, `VS_likelihood` and `expected_lifetime`. These are discussed in detail in the advanced usage example in Sect. 2.5.2.

2.2.4 MSSM

The **capabilities** available in SpecBit relevant for the MSSM model are given in Table 2.

SpecBit has a number of module functions that provide the **capability** `unimproved_MSSM_spectrum`. We will start by examining three of them:

    `get_MSSMatMGUT_spectrum`,
    `get_MSSMatQ_spectrum`, and
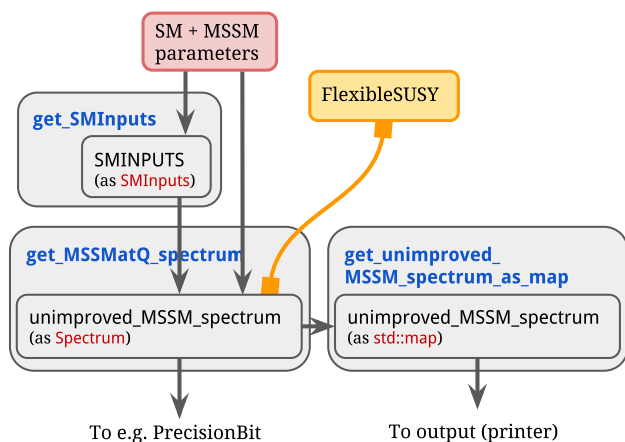   `get_CMSSM_spectrum`.

Fig. 2 A typical flow of information through SpecBit, using MSSM module functions as examples. First, model parameters (red box) are obtained from the GAMBIT Core. Some of these are used to construct Standard Model input information for the spectrum generation (`SMINPUTS`, upper grey box), while MSSM parameters are passed directly to the spectrum builder function (`get_MSSMatQ_spectrum`, lower left grey box). This function constructs a `Spectrum` object, which wraps information obtained from FlexibleSUSY (yellow box), and passes it on to other parts of GAMBIT. In addition, the `Spectrum` object is translated into a C++ map (lower right grey box), which can be parsed by the GAMBIT printer system and written to disk

These module functions provide the `unimproved_MSSM_spectrum` **capability** in the form of a `Spectrum` object by calling one of the FlexibleSUSY spectrum generators described in Sect. 2.1.2. The module functions all depend on `SMINPUTS` provided in the form of an `SMInputs` structure. These spectra are of the most sophisticated type, providing full two-loop RGE running, as described in Sect. 2.3.3. Options can be supplied to FlexibleSUSY via the input YAML file, and they are discussed in Sect. 2.2.5.

Additionally the module function

```
get_MSSM_spectrum_SPheno
```

also provides the **capability** `unimproved_MSSM_spectrum`, but in this case obtains the information from a different spectrum generator: SPheno. As with the module functions that call FlexibleSUSY, this module function depends on an `SMInputs` object and provides a `Spectrum` object, selecting the specific model and running mode of the backended SPheno at runtime. In contrast to the FlexibleSUSY case, however, this `Spectrum` object does not provide RGE running. Is is simply a static spectrum equivalent to the contents of an SLHAea object (`SLHAstruct`), which is a simple container of SLHA-like information (described in more detail in Sect. 2.3.3 under *SLHAea output*). The options available for the backend version of SPheno used by this module function are in Sect. 2.2.6.

The module function

```
get_MSSM_spectrum_from_SLHAfile
```

also provides this `unimproved_MSSM_spectrum` **capability** in the form a `Spectrum` object. However, instead of using a spectrum generator to do this it uses an SLHA file. The `Spectrum` object in this case does not provide the ability for RGE evolution, but is rather a simple container spectrum. This function takes the options

- `filenames`: path to SLHA file to be read in. Can be a list of many file names, in which case each file will be read in sequentially.
- `cycles`: number of loops over the `filenames` list to allow (default `-1` indicates no limit). When the limit is hit an error will be raised, stopping the scan.

This function is mainly for debugging purposes, because one usually generates spectra on the fly during a scan.

Two further SpecBit module functions,

```
get_unimproved_MSSM_spectrum_as_SLHAea and
get_unimproved_MSSM_spectrum_as_map
```

are available for translating a `Spectrum` object to either an SLHAea object (`SLHAstruct`) or to a C++ `map` (as in the analogous scalar-singlet DM function), respectively. These still represent the spectrum, just with a different type, so the module functions have both a **dependency** and a **capability** of `unimproved_MSSM_spectrum`, but with different types. See Fig. 2 for an example use case of this 'translation' behaviour. We have found each format useful for different tasks. For example, some module functions in other parts of GAMBIT request spectrum information in `SLHAstruct` format, rather than as full `Spectrum` objects, because these functions work closely with **backend** codes that were originally designed to work via SLHA files. So keeping the format directly parallel to SLHA was convenient for them. On the other hand, the `std::map` format is useful because at present neither `Spectrum` objects nor `SLHAstruct` objects can be directly written to disk by the GAMBIT printer system [11], so converting to `std::map` is necessary as an intermediate step for this purpose.

The module functions

```
get_MSSM_spectrum_as_SLHAea and
get_MSSM_spectrum_as_map
```

also translate a `Spectrum` object into an SLHAea structure or a `std::map`. However, in these cases the **capability** and **dependency** is `MSSM_spectrum`, a **capability** that can also be provided by the module functions from PrecisionBit, as shown in module function Table 10 and discussed in Sect. 4.2.3.

Table 2 MSSM **capabilities** provided by SpecBit. The **capabilities** `MSSM_spectrum` and `unimproved_MSSM_spectrum` are understood to provide MSSM couplings and soft masses in the $\overline{DR}$ scheme, along with pole masses, with low-energy Standard Model inputs defined in the $\overline{MS}$ scheme. Note also that module functions exist in PrecisionBit with **capability** `MSSM_spectrum`, which depend on the `unimproved_MSSM_spectrum`; see Table 10 and Sect. 4.2.3 for more details

Capability	Function (return type): brief description	Dependencies	Backend requirements
`unimproved_MSSM_spectrum`	`get_CMSSM_spectrum(Spectrum)`: Make an MSSM spectrum object with FlexibleSUSY using CMSSM boundary conditions	`SMINPUTS`	FlexibleSUSY
	`get_MSSMatMGUT_spectrum(Spectrum)`: Make an MSSM spectrum object with FlexibleSUSY using soft-breaking masses input at the GUT scale	`SMINPUTS`	FlexibleSUSY
	`get_MSSMatQ_spectrum(Spectrum)`: Make an MSSM spectrum object with FlexibleSUSY from soft-breaking masses input at (user-defined) scale Q	`SMINPUTS`	FlexibleSUSY
	`get_MSSM_spectrum_SPheno(Spectrum)`: Make an MSSM spectrum object with SPheno for any supported MSSM model	`SMINPUTS`	SPheno
	`get_MSSM_spectrum_from_SLHAfile(Spectrum)`: Construct an MSSM spectrum from an SLHA file. Wraps it up in `MSSMSimpleSpec`; i.e. no RGE running possible. Designed for testing against benchmark points, but may be a useful last resort for interacting with "difficult" spectrum generators		
	`get_unimproved_MSSM_spectrum_as_SLHAea(SLHAstruct)`: Convert an unimproved MSSM spectrum into an `SLHAstruct` object	`unimproved_MSSM_spectrum`	
	`get_unimproved_MSSM_spectrum_as_map(map_str_dbl)`: Convert an unimproved MSSM spectrum into a `std::map`, for saving via the GAMBIT printer system	`unimproved_MSSM_spectrum`	
`MSSM_spectrum`	`get_MSSM_spectrum_as_map(map_str_dbl)`: Convert an MSSM spectrum into a `std::map`, for saving via the GAMBIT printer system	`MSSM_spectrum`	
	`get_MSSM_spectrum_as_SLHAea(SLHAstruct)`: Turn an MSSM spectrum into an `SLHAstruct` object	`MSSM_spectrum`	
`SM_subspectrum`	`get_SM_SubSpectrum_from_MSSM_Spectrum(const SubSpectrum*)`: Extract the "low-energy" SubSpectrum from an MSSM Spectrum object	`MSSM_spectrum`	
`SMlike_Higgs_PDG_code`	`most_SMlike_Higgs_MSSM(int)`: Return the PDG code of the most SM-like of the neutral Higgs bosons in the MSSM	`MSSM_spectrum`	

Table 2 continued

Capability	Function (return type): brief description	Dependencies	Backend requirements
Higgs_Couplings	MSSM_higgs_couplings_pwid (HiggsCouplingsTable): Construct a table of Higgs couplings from Higgs partial decay widths	MSSM_spectrum, SMlike_Higgs_PDG_code, Reference_SM_Higgs_decay_rates, Reference_SM_other_Higgs_decay_rates, Reference_SM_A0_decay_rates, Higgs_decay_rates, h0_2_decay_rates, A0_decay_rates, H_plus_decay_rates, t_decay_rates	
	MSSM_higgs_couplings_FH (HiggsCouplingsTable): Construct a table of Higgs couplings from couplings calculated by FeynHiggs	MSSM_spectrum, SMlike_Higgs_PDG_code, Reference_SM_Higgs_decay_rates, Reference_SM_other_Higgs_decay_rates, Reference_SM_A0_decay_rates, Higgs_decay_rates, h0_2_decay_rates, A0_decay_rates, H_plus_decay_rates, t_decay_rates, FH_Couplings_output	
FH_Couplings	FH_Couplings_output (fh_Couplings): Higgs couplings, as computed by FeynHiggs	prec_HiggsMasses	FeynHiggs
prec_HiggsMasses	FH_HiggsMasses (fh_HiggsMassObs): Higgs masses and mixings with theoretical uncertainties, as computed by FeynHiggs	FH_MSSMMasses	FeynHiggs
FH_MSSMMasses	FH_MSSMMasses (fh_MSSMMassObs): SUSY masses and mixings, as computed by FeynHiggs		FeynHiggs

In the MSSM it is often useful to be able to identify which Higgs boson is most similar to the SM Higgs boson. The module function

```
most_SMlike_Higgs_MSSM
```

provides this information by returning a PDG code representing the CP-even Higgs with couplings that are closest to those of the SM Higgs boson. We deem the lightest CP-even state to be the most SM-like if $\sin(\beta-\alpha) > \cos(\beta-\alpha)$, where α is the mixing angle defined in Eq. (A.16); otherwise, this function returns the PDG code of the heavier CP-even Higgs. This function has **capability** `SMlike_Higgs_PDG_code`, and a **dependency** on the `MSSM_spectrum`.

It is also possible to extract the low-energy SM spectrum from the MSSM spectrum. The **capability** `SM_subspectrum` is provided by the module function

```
get_SM_SubSpectrum_from_MSSM_Spectrum,
```

which has a **dependency** on the `MSSM_spectrum`.

Finally, there are three module functions that require backend functions from FeynHiggs. These module functions,

```
FH_MSSMMasses,
FH_HiggsMasses and
FH_Couplings,
```

respectively, provide MSSM masses with **capability** `FH_MSSMMasses`, Higgs masses with capability `prec_HiggsMasses`, and Higgs couplings with capability `FH_Couplings_output`.

2.2.5 FlexibleSUSY options

Any spectrum generator interfaced to GAMBIT will inevitably have its own set of options to control precision and methods of calculation. Here we briefly summarise the FlexibleSUSY options, which can be set via YAML options. For a more detailed explanation of these options, see the FlexibleSUSY manual [19].

– `precision_goal`: relative error for the calculation of the $\overline{DR}$ spectrum. Default 1×10^{-4}.
– `max_iterations`: maximum number of iterations for the two-scale algorithm. Default $-10 \log_{10}($`precision_goal`$)$.
– `calculate_sm_masses`: compute SM pole masses during spectrum calculation. Default **false**.
– `pole_mass_loop_order`: number of loops in self energies for pole mass calculation. Default 2.

– `ewsb_loop_order`: number of loops in radiative corrections to the EWSB conditions in Eq. 33 of Ref. [19]. Default 2.
– `beta_loop_order`: loop order for RGEs. Default 2.
– `threshold_corrections_loop_order`: loop order for threshold corrections as defined in Eqs. 6–13 of Ref. [19]. Default 2.
– Two-loop Higgs pole mass corrections are controlled for specific terms with a boolean input, these are by default all set to **true**. The available terms are:

   – `use_higgs_2loop_at_as`: $\mathcal{O}(\alpha_t \alpha_s)$,
   – `use_higgs_2loop_ab_as`: $\mathcal{O}(\alpha_b \alpha_s)$,
   – `use_higgs_2loop_at_at`: $\mathcal{O}(\alpha_t^2 + \alpha_t \alpha_b + \alpha_b^2)$,
   – `use_higgs_2loop_atau_atau`: $\mathcal{O}(\alpha_\tau^2)$,

where for $x = t, b, \tau$, $\alpha_x = y_x^2/(4\pi)$ and y_x is the corresponding Yukawa coupling, and α_s is the strong coupling constant.

Note that the included spectrum generators created by FlexibleSUSY perform a fixed-order calculation of the Higgs mass and we do not currently support additional FlexibleSUSY codes[9] HSSUSY and FlexibleEFTHiggs that resum large logarithms. However, we refer the reader to Ref. [55] for a discussion of how the large logarithms fortuitously cancel in the fixed-order FlexibleSUSY calculation, such that the uncertainty is much lower at higher scales in the MSSM than one would naively expect.

If the FlexibleSUSY spectrum generator encounters an error during calculation, this may be passed on as a SpecBit error, resulting in the termination of a scan, or as an invalid point exception, which will result in the scan point being given an invalid likelihood (an extremely small value defined by `model_invalid_for_lnlike_below` in the YAML file). See Ref. [11] for more details on exception handling. The management of these exceptions within GAMBIT is controlled with the option `invalid_point_fatal` discussed in Sect. 2.2.3.

2.2.6 SPheno options

As with FlexibleSUSY, there are a number of options available for SPheno to control certain aspects of the calculation. In GAMBIT only a limited set of all the available options for SPheno is allowed, because many features or models permitted by SPheno are not yet covered in GAMBIT. A detailed explanation of all the options available for SPheno can be found in the manual [21], under the SLHA block SPhenoInput.

---

[9] Such options will be added at a later date, however, in the intervening period, an advanced user may also add these codes themselves, following the instructions given in this manual.

The following options can be provided via the YAML options section in the input YAML file, and they will be internally assigned to the corresponding variables in SPheno.

- `SPA_convention`: use the SPA convention [65], effectively setting the parameter output scale to 1 TeV. Default `false`.
- `GUTScale`: fixed value for the GUT scale. Default 0.0, which causes the GUT scale to be obtained at runtime.
- `StrictUnification`: force strict unification of the gauge couplings $g_1 = g_2 = g_3$. Default `false`.
- `delta_mass`: relative precision in the calculation of the masses. Default 10^{-6}.
- `n_run`: maximum number of iterations for the mass calculation. Default 40.
- `TwoLoopRGE`: whether to use two-loop RGEs. Default `true`.
- `Alpha`: value of the fine-structure constant $\alpha(0)$. Default $\frac{1}{137}$.
- `gamZ`: value of the decay width of the Z gauge boson. Default 2.49 GeV.
- `gamW`: value of the decay width of the W gauge boson. Default 2.04 GeV.
- `Use_bsstep_instead_of_rkqs`: use the `bsstep` algorithm instead of `rkqs`. Default `false`.
- `Use_rzextr_instead_of_pzextr`: use the `rzextr` algorithm instead of `pzextr`. Default `false`.

Note that the MSSM SPheno spectrum generator supported currently is the one immediately available after downloading SPheno, rather than including auto-generated code from SARAH. It includes only the fixed-order Higgs-mass calculation, so the hybrid EFT/fixed-order calculation based on FlexibleEFTHiggs [55] which has been developed very recently [66] is not currently available in SpecBit.

Errors prompted by SPheno are collected by the internal integer variable `kont`, which takes different values according to the source of the error. The specific error messages corresponding to values of `kont` can be found in Appendix C of the manual [21]. For any of these values, the backend convenience function `run_SPheno` raises an `invalid_point` exception, as described in [11].

2.2.7 Mass cut options

For any module function that constructs a `Spectrum` object, it is possible to specify options that enforce user-defined relationships between particle pole masses: `mass_cut` and `mass_cut_ratio`. Any spectrum that does not pass the specified cuts is declared invalid (which may invalidate the entire parameter point, depending on whether the likelihood ultimately depends on the spectrum or not [11]). While these cuts are not, in general, physically required, we have found

this feature useful for certain specialised scans, for example explorations of co-annihilation regions in the MSSM.

These options take one or two particle names recognised by the spectrum, along with two numbers used to define the range to be cut. They are best explained by way of an example entry in the `Rules` section of a GAMBIT YAML file:

```
- capability: unimproved_MSSM_spectrum
  function: get_CMSSM_spectrum
  options:
    mass_cut: [[
"h0_1",120,130],["~e-_1",100,1000]]
    mass_ratio_cut: [[
"~e-_1","|~chi0_1|",1,1.01]]
```

The above code invalidates any points that do not have:

- the mass of the lightest Higgs between 120 and 130 GeV,
- the mass of the lightest slepton between 100 and 1000 GeV, and
- the mass of the lightest slepton within 1% of the mass of the lightest neutralino.

Note in particular the use of absolute value signs for the neutralino mass, which may be negative. This notation can be used wherever a particle name is given, in order to define the cut rule in terms of the absolute value of the particle mass. Also note that two such cut rules are given for `mass_cut`, illustrating the fact that these options can be used to apply arbitrarily many cuts simultaneously.

2.3 Interface details for GAMBIT module writers (C++ API for `Spectrum` and related classes)

In Sect. 2.2 we provided details for a base-level user of GAMBIT to simply operate the code as written. However, one of the goals of GAMBIT is to provide a framework into which researchers can add their own calculations while maintaining easy use of all pre-existing GAMBIT functions. This section is intended as a guide for users of this kind. This information will also be needed by users running SpecBit externally to GAMBIT. Here we describe details of the interfaces to objects provided by the SpecBit module, and how they should be used to facilitate physics calculations of other modules.

In Sect. 2.3.1 we demonstrate the most common methods for accessing spectrum information. In Sects. 2.3.2 and 2.3.3, we outline the class structure used to store spectrum information, and explain the details of its interface. The helper class `SMInputs` is discussed last, in Sect. 2.3.5.

2.3.1 Basic spectrum access

The spectrum information is stored in a `Spectrum` object, the full structure of which will be described in Sect. 2.3.2. However, here we briefly describe the basic user interface.

To access a `Spectrum` object in a new module function, the module function should be declared to have a **dependency** on the appropriate spectrum. As described in section 3.2.1 of the GAMBIT paper [11], the spectrum object can then be obtained by dereferencing the safe pointer returned by the SpecBit module function, e.g.

```
using namespace Pipes::my_module_function;
const Spectrum& spec =
    *Dep::SingletDM_spectrum;
```

One can now access the spectrum information using `spec`. The spectrum information is primarily accessed using a string accessor plus a tag, as in

```
double mh0 =
    spec.get(Par::Pole_Mass,"h0_1");
```

Note that `Par` is simply a namespace, so the tags can be brought into scope with the `using` keyword, and then used more succinctly, i.e.

```
using namespace Par; double mh0 =
    spec.get(Pole_Mass,"h0_1");
```

The tag in the first argument of the getter specifies the type of information to look for. The second argument is a string that, in combination with the tag, tells the getter exactly what information is requested. To simplify access to quantities that are naturally grouped together in vectors (e.g. CP-even Higgs or neutralino masses in the MSSM), the `get` function may also be called with an index:

```
double mh0 = spec.get(Pole_Mass,"h0",1);
```

As well as a `get` function, `Spectrum` objects contain a `has` function with an almost identical function signature, which can be used to check if a quantity exists in the spectrum before attempting to retrieve it:

```
bool has_mh0_1 = spec.has(Pole_Mass,"h0",1);
```

where `has_mh0_1` is `true` if the quantity exists in the spectrum, and `false` if it does not.

It is also possible to `set` the values of spectrum parameters, but not via the base-level `Spectrum` object interface. For this more advanced usage, see Sect. 2.3.3. Most users will not need to do this.

This form of string getter allows the spectrum information to be accessed in a simple, uniform way across all models and all GAMBIT modules. The user simply needs to know the specific tags and string names used for each piece of information. Particle names, for example, can be found in the GAMBIT particle database (see Sect. 10.1 of Ref. [11]).
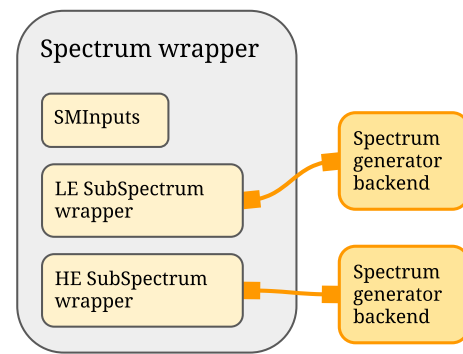

Fig. 3 Schematic representation of a `Spectrum` object. Basic Standard Model "input" information is contained in a structure call `SMInputs` (upper-left light-yellow box), the content of which mirrors the correspondingly named SLHA2 block. Low scale (generally $< m_Z$) spectrum information is wrapped by a "low-energy" `SubSpectrum` object (centre-left light-yellow box). High scale (generally $> m_Z$) spectrum information is wrapped in a "high-energy" `SubSpectrum` object (lower-left light-yellow box). This split mirrors a common requirement of spectrum generators that Standard Model input parameters be first supplied at a common scale, then run by the "low-energy" `SubSpectrum` object to a matching scale, after which the main spectrum generation for the BSM model is performed and wrapped into the "high-energy" `SubSpectrum` object. The outer `Spectrum` object provides a uniform interface to these underlying structures and makes it easy to retrieve common parameters such as pole masses without the need to interact directly with the subspectra

For convenience, in Appendix C we give tables listing the tags, strings, and indices needed to access data via the getters, checkers, and setters for the SM, the MSSM, and the scalar-singlet model.

Note that only information tagged with `Pole_Mass` can be retrieved via the base-level `Spectrum` interface. For other information (and to run RGEs, when available) one must use the equivalent `SubSpectrum` interface, which is described in Sect. 2.3.3.

2.3.2 *Spectrum class structure*

At the centre of the SpecBit module is a virtual interface class named `Spectrum`, for accessing typical spectrum generator output in a generalised and standardised way. Wherever possible, SLHA2 conventions [1,2] are used as the standard. The objects accessed by this interface can thus be considered as an in-memory representation of the data that one would typically find in the SLHA2 ASCII output of a spectrum generator.

Each `Spectrum` object contains three main data members, illustrated in Fig. 3: an `SMInputs` object, which contains SM input parameters (closely mirroring the `SMINPUTS` block defined by SLHA2); and two `SubSpectrum` objects, one labelled `LE` for 'low energy' and which typically contains low-scale SM information, and the other labelled `HE` for 'high energy', which typically contains higher-scale model infor-

```cpp
class Spectrum
{
  public:
    SubSpectrum& get_LE();        // Access "low-energy" SubSpectrum
    SubSpectrum& get_HE();        // Access "high-energy" SubSpectrum
    SMInputs&    get_SMInputs();  // Access SMInputs associated with this Spectrum

    std::unique_ptr<SubSpectrum> clone_LE() const; // Copy "low-energy" SubSpectrum
    std::unique_ptr<SubSpectrum> clone_HE() const; // Copy "high-energy" SubSpectrum
    // (whole Spectrum object can be copied using standing copy constructor/assignment methods)

    void RunBothToScale(double scale); // Run both LE and HE SubSpectrum to same scale
                                       // (Default units: GeV)
    // Get/check/set functions for parameter retrieval/input with zero or one indices.
    bool   has(const Par::Tags partype, const std::string& mass, const int index=NONE) const;
    double get(const Par::Tags partype, const std::string& mass, const int index=NONE) const;

    SLHAstruct getSLHAea(int slha_version) const; // Create SLHAea object from LE+HE SubSpectra in
  SLHA standard slha_version. HE takes precedence.
    void writeSLHAfile(int slha_version, str fname) const; // Output spectrum contents as an SLHA
  file, conforming to SLHA standard slha_version.
};
```

Fig. 4 Simplified class declaration of primary `Spectrum` class. Not shown are various overloads, constructors and assignment operators, helper functions, and all private members. Additionally, the index arguments for the has/get functions are implemented via operator overloads rather than optional arguments as is shown. For the full class declaration please see `Elements/include/gambit/Elements/spectrum.hpp` in the **GAMBIT** source code

mation such as Higgs-sector or BSM data. The reason for this separation is that, typically, one requires different calculations for accurately evaluating and evolving low-energy SM parameters (like quark masses) and high-energy ones like the MSSM. The code to do each calculation will be wrapped in a separate `SubSpectrum` structure. This wrapper also provides access to running parameters and RGE-running facilities, if these are available in a backend code wrapped by a particular `SubSpectrum`. We describe the `SubSpectrum` interface in Sect. 2.3.3. Details needed for writing such a class are found in Sect. 2.4. The purpose of the host `Spectrum` object is to provide a consistent interface to the underlying subspectra and SM input parameters.

In Fig. 4 we provide a simplified overview of the contents of the `Spectrum` class, for reference. We will now discuss the various functions and data members shown in this figure.

Data member getters The central members of a `Spectrum` object are accessed via three functions:

`get_LE()`/`get_HE()` Accesses the hosted `SubSpectrum` object identified as 'low energy' or 'high energy'.
`get_SMInputs()` Accesses the hosted `SMInputs` object.
`clone_LE()`/`clone_HE()` Creates a copy of the hosted `SubSpectrum` object identified as 'low energy' or 'high energy'.

The 'clone' functions exist because by design, it is not possible to perform actions that modify contents of `Spectrum` objects provided to **GAMBIT** module functions as dependen-

cies. This protection extends to the `SubSpectrum` objects, so in order to perform actions that modify the spectrum (like RGE running) one must first copy the whole `Spectrum` or 'clone' the relevant `SubSpectrum`.

Copying `Spectrum` objects Because **GAMBIT** prevents the results of module functions from being modified by other module functions, if a user wishes to modify the contents of a **SpecBit**-provided `Spectrum` object, the user must first copy it. The copy constructor (and associated constructors) of the `Spectrum` object are designed to perform a 'deep' copy. To make a copy, one can simply use the copy constructor or assignment operators, i.e.

```cpp
Spectrum spec_copy(spec);//This is a deep copy
Spectrum spec_copy=spec; //Also a deep copy
```

RGE running There is little in the underlying object that can be modified from the `Spectrum` interface, except via the `RunBothToScale` function. This function runs the renormalisation-scale-dependent parameters of the underlying `SubSpectrum` objects to the same scale. Note that the underlying RGEs may not be valid beyond certain scales on both the high and low ends, so some caution and knowledge of the underlying objects is required in order to safely use this feature (see 'RGE running' in the following subsection).
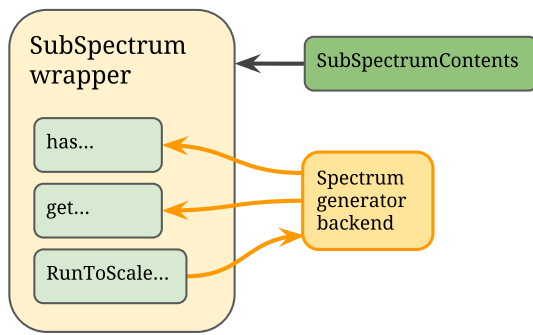
Fig. 5 Schematic representation of a `SubSpectrum` object. These objects interface directly to spectrum information or external spectrum generators (yellow box). Accessor functions (e.g. `has`, `get`) are used to access spectrum parameters, and, in the case where a spectrum generator is connected, the `RunToScale` function can be called to run the whole spectrum to a different scale (pale-green boxes). To standardise parameters that should be retrievable from generated spectra for each model, we define a separate `SubSpectrumContents` object for each model. Each wrapper is then associated with some `SubSpectrumContents` (green box), and is required to provide all the contents defined therein

2.3.3 *SubSpectrum objects*

While the `Spectrum` interface provides fast and easy access to pole masses, it does not allow access to more detailed spectrum information, such as the values of couplings, mixings, and running mass parameters. For this purpose one must use the `SubSpectrum` interface (Fig. 5). As we covered in Sect. 2.3.2, the simplest way to obtain a `SubSpectrum` object is to extract it from a host `Spectrum` object using the `get_LE()`/`get_HE()` functions or `clone_LE()`/`clone_HE()` functions:

```
const SubSpectrum& mssm = spec.get_HE();
unique_ptr<SubSpectrum> copy = spec.clone_HE();
```

Following the same pattern as the `Spectrum` interface, spectrum information can be accessed via the `SubSpectrum` interface using a `get` function, for which the arguments are a tag, a string identifier, and zero to two indices:

```
using namespace Par;
double mass_su1 = mssm.get(Pole_Mass,"~u",1);
```

Let us now examine the contents of the `SubSpectrum` interface class. This class is a virtual interface for derived classes individually designed to wrap the data from external spectrum generator codes. The virtualisation allows module writers to treat spectrum data in their own module functions in a generic way, with the selection of the actual spectrum generator to use being a decision left to runtime. The wrapping of external codes is then decoupled from the use of the data required by module functions, meaning that replacement wrappers can be written for new external spectrum generator codes, and the new wrapper activated, without necessitating any modification of downstream module functions.

For reference, a simplified outline of the `SubSpectrum` class is shown in Fig. 6. For the precise class declaration please see the file

```
Elements/include/gambit/Elements/subspectrum.
hpp
```

in the GAMBIT source tree.

Getters, checkers and setters Like the `Spectrum` interface, the `SubSpectrum` interface also provides a `has` function, which accepts the same input arguments as the `get` function, and which returns a `bool` that indicates whether or not a given `SubSpectrum` object contains the requested parameter:

```
bool has_mass_su1 = mssm.has(Pole_Mass,"~u",1);
```

Unlike the `Spectrum` interface, the `SubSpectrum` interface also provides `set` functions. These can be used to modify or overwrite parameter data contained in the `SubSpectrum` object. For example,

```
double newmass  = 500; // Units generally in
double newmass2 = 600; // powers of GeV
mssm.set(Pole_Mass,newmass,"~u",1);
mssm.set_override(Pole_Mass,newmass2,"~u",1);
```

The `set` and the `set_override` functions differ in an important way. The `set` function, if permitted by the underlying wrapper class, will directly change a value in the underlying spectrum generator code, and thus may, for example, impact renormalisation group running. On the other hand, the `set_override` function does not see the underlying spectrum generator code at all, and it will simply store a replacement value that will be preferentially retrieved when using the `get` function. This can be useful for storing, for example, precision calculations of pole masses in such a way that they can be seamlessly used by other GAMBIT module functions, via the standard interface. We emphasise that the `set_override` value will take ultimate priority, and in fact the underlying value known to the spectrum generator code will become masked, and irretrievable. So in the example above, a call to `mssm.get(Pole_Mass,"~u",1)` after the call to `mssm.set_override` would retrieve the value 600.

There is also an optional final `bool` argument to `set_override` (not shown in Fig. 6), which indicates whether the function is permitted to add to the contents of a `SubSpectrum`. The default is `false`, leading to an error if a user tries to override a part of the `SubSpectrum` contents that does not already exist. By setting this `true`, one can insert arbitrary new information into the `SubSpectrum`, which will become retrievable with the `get` method. This functionality should be used sparingly, as the writers of other module functions generally will not be aware of the existence of the

```cpp
class SubSpectrum
{
  public:
    std::unique_ptr<SubSpectrum> clone() const;   // Clone (copy) this SubSpectrum object

    virtual void writeSLHAfile(int slha_version, const str& fname) const; // Write out spectrum to SLHA
  ↪file (if possible) in SLHA standard slha_version
    virtual SLHAstruct getSLHAea(int slha_version) const;          // Get spectrum in SLHAea format (if
  ↪possible)
    virtual void add_to_SLHAea(int slha_version,SLHAstruct&) const;// Add spectrum data to an SLHAea
  ↪object (if possible) in SLHA standard slha_version

    // Limits to RGE running; warning/error raised if running beyond these is attempted.
    // If these aren't overridden in the derived class then effectively no limit on running will exist.
    // These are public so that module writers can use them to check what the limits are.
    virtual double hard_upper() const {return DBL_MAX;}
    virtual double soft_upper() const {return DBL_MAX;}
    virtual double soft_lower() const {return 0.;}
    virtual double hard_lower() const {return 0.;}
    // Returns the renormalisation scale of parameters. By default should be in units of GeV.
    virtual double GetScale() const;
    void RunToScale(double scale, int behaviour = 0); // Run spectrum to scale (Default units: GeV).

    // Get/check/set functions for parameter retrieval/input with zero, one, and two indices
    virtual bool    has(const Par::Tags, const str&, int index1=NONE, int index2=NONE) const;
    virtual double  get(const Par::Tags, const str&, int index1=NONE, int index2=NONE) const;
    virtual void    set(const Par::Tags, const double, const str&, int index1=NONE, int index2=NONE);
    void  set_override(const Par::Tags, const double, const str&, int index1=NONE, int index2=NONE);
};
```

Fig. 6 Simplified class definition of `SubSpectrum` class, which is the virtual base class for interacting with spectrum generator output, including an interface for running RGEs. Not shown are various overloads, constructors, helper functions, and all private members. The index arguments for the `has`/`get`/`set` func-

tions are implemented via operator overloads rather than the optional arguments shown. For the full class definition please see `Elements/include/gambit/Elements/subspectrum.hpp` in the GAMBIT source code

additional data. If you find yourself wanting to add large amounts of new data to a spectrum, it is generally better to define a new `SpectrumContents` and to create a new spectrum wrapper class that conforms to this expanded contents set (see Appendix F.1).

RGE running The `SubSpectrum` interface provides the main connection to RGE-running facilities in any underlying spectrum-generator code; the `RunBothToScale` function of the `Spectrum` interface is just a thin wrapper over this. These facilities do not exist in all `SubSpectrum` objects, so one must make sure, via the GAMBIT **dependency resolution** process, that an appropriate spectrum is supplied to module functions that require running facilities. If a module function attempts to call any of the following functions when no underlying RGEs are connected, then SpecBit will throw an error and terminate.

The two most fundamental RGE-related functions are the `RunToScale` and `GetScale` functions. The latter function takes no arguments and simply returns the energy scale at which any running parameters accessible by the `get` function are defined, for example couplings and running masses. The `RunToScale` function accepts a new energy scale as input,

and when called will run the underlying RGE code and recalculate the running parameters.

The precise behaviour of the `RunToScale` function can be altered via the optional integer '`behaviour`' argument. This controls how attempts to run parameters beyond the known accurate range of the underlying RGEs are handled. There exist two sets of limits in the `SubSpectrum` interface; upper and lower "soft" limits, and upper and lower "hard" limits. These specify the energy ranges that affect the action of the `behaviour` switch, and can be checked with the following functions:

```cpp
virtual double hard_upper() const
virtual double soft_upper() const
virtual double soft_lower() const
virtual double hard_lower() const
```

Note that when writing a new `SubSpectrum` wrapper these functions need to be overridden to specify valid ranges of RGE evolution, otherwise the default limits (which effectively allow unlimited running) will be used.

The "behaviour" flag acts in conjunction with the hard and soft limits according to the following rules:

- `behaviour = 0` (default) – If running beyond soft limit requested, halt at soft limit (parameter evolution will simply stop silently at this limit).
- `behaviour = 1` – If running beyond soft limit requested, throw warning; beyond hard limit, throw error.
- `behaviour = <anything else>` – Ignore limits and attempt running to requested scale (errors may still be thrown by the backend spectrum generator code if it sets its own hard-coded limits).

SLHAea output Strictly speaking, the `SubSpectrum` objects do not rely on SLHA, and can interface to the spectrum codes for any model. However, SLHA is an existing standard for the MSSM, so we provide the spectrum in this standard. The `SubSpectrum` interface thus provides three functions that convert `SubSpectrum` contents into SLHA2 format, making use of the **SLHAea** C++ library:[10]

- `void writeSLHAfile(int slha_version,const str& fname)` Writes out the contents of the `SubSpectrum` in SLHA format with the name `fname`. If `slha_version` is set to 1 then it will conform to SLHA1 standard, while if it is set to 2 then it will conform to the SLHA2 standard. Ideally this should only be used for debugging purposes, as minimising disk access is preferable in high-performance computing applications. However, some backend codes do not have proper APIs and can only be run via the command line, or have otherwise hard-coded requirements for input SLHA files.
- `SLHAstruct getSLHAea(int slha_version)` Writes the contents of the `SubSpectrum` into an `SLHAea::Coll` object (for which `SLHAstruct` is an alias). This is an in-memory string-based representation of an SLHA file, which is in the SLHA standard `SLHA_version` while negating any need to write to disk. Block contents can easily be accessed via `[]` operators (see the **SLHAea** documentation for examples).
- `void add_to_SLHAea(SLHAstruct&)` Similar to `getSLHAea`, however, the spectrum content is added to a pre-existing `SLHAstruct`, passed in by reference via the input argument and modified in place.

SLHAea is very flexible and can be used to also create SLHA-like structures for other models. Therefore this may also be exploited for non-minimal SUSY models and non-SUSY models too, though other than the MSSM and NMSSM there is no existing standard to fix the precise form and ensure that other codes will understand the results. However, this feature

is also rather useful for auto-generated programs, where one can construct the interface between the different generated codes to work with new SLHA blocks.

2.3.4 The `SMInputs` class

The three data members of a `Spectrum` object are two `SubSpectrum` objects, plus an `SMInputs` object. The latter object does not strictly contain information about the calculated spectrum, and the data it contains is not accessed via the `get` function of the host `Spectrum` object. Rather, it contains information that in most cases was used as *input* to the external spectrum generator code. It is a simple `struct` that directly mirrors the SMINPUTS, VCKMIN and UPMNSIN blocks defined by the SLHA2 standard (see Fig. 7).

Because `SMInputs` is a structure, all data members (the elements of the SLHA2 blocks) have public access, and so parameter values can be accessed directly (see Fig. 7 for the variable names). The structure contains only two functions, `getSLHAea` and `add_to_SLHAea`. The former simply returns the contents of the structure as an SLHAea object (which can be easily written to disk, for example), while the second adds the contents of the structure to an existing SLHAea object. The latter feature is mainly used internally by `Spectrum` objects to construct a full SLHA2 representation of their contents.

2.3.5 Extra overloads for `get`/`set`/`has` functions

In the discussions of the interface for `Spectrum` and `SubSpectrum` objects, we have mentioned only one set of function signatures for the `get`, `set` and `has` functions used to access spectrum data, however, several overloads of these functions are available. Most of these are designed to streamline the interaction between `Spectrum` objects and the **GAMBIT** particle database (see Sect. 10.1 of Ref. [11]), and they allow particle data to be retrieved according to PDG codes and, where appropriate, antiparticle string names (see `Models/src/particle_database.cpp` for the full set of definitions known to **GAMBIT**).

For `Spectrum` objects, the extra function signatures are as follows:

```
// PDG code + context integer
bool   has(tag, PDGcode, context)
double get(tag, PDGcode, context)

// PDG code + context integer (as std::pair)
bool   has(tag, std::pair<int,int>)
double get(tag, std::pair<int,int>)

// Short name plus index (as std::pair)
bool   has(tag, const std::pair<str,int>)
double get(tag, const std::pair<str,int>)
```

---

[10] http://fthomas.github.io/slhaea/.

```
struct SMInputs
{
    // SLHA1
    double alphainv;   // 1:
    double GF;         // 2:
    double alphaS;     // 3:
    double mZ;         // 4:
    double mBmB;       // 5:
    double mT;         // 6:
    double mTau;       // 7:
    // SLHA2
    double mNu3;       // 8:
    // double mW;      // 9:
    double mE;         // 11:
    double mNu1;       // 12:
    double mMu;        // 13:
    double mNu2;       // 14:

    double mD;         // 21:
    double mU;         // 22:
    double mS;         // 23:
    double mCmC;       // 24:

    // Block VCKMIN
    struct CKMdef {
      double lambda;
      double A;
      double rhobar;
      double etabar; };
    CKMdef CKM;

    // Block UPMNSIN
    struct PMNSdef {
      double theta12; //
      double theta23; //
      double theta13; //
      double delta13; //
      double alpha1;  //
      double alpha2;  //
    };
    PMNSdef PMNS;

    // Retrieve contents as an SLHAea object
    SLHAstruct getSLHAea(int getSLHAea) const;

    // Add contents to existing SLHAea object;
      void add_to_SLHAea(int
↪slha_version,SLHAstruct& slha) const;
    };
```

Fig. 7 Definition of the `SMInputs` class. No simplification has been performed; this is the full class definition as found in `Elements/include/gambit/Elements/sminputs.hpp` (with condensed comments)

For `SubSpectrum` objects the above extra function signatures also exist, plus matching `set` and `set_override` functions:

```
// PDG code + context integer
void    set(tag, value, PDGcode, context)
void    set_override(tag, value, PDGcode, context)

// PDG code + context integer (as std::pair)
void    set(tag, value, std::pair<int,int>)
void    set_override(tag, value,
   std::pair<int,int>)

// Short name plus index (as std::pair)
```

```
void    set(tag, value, const
   std::pair<str,int>)
void    set_override(tag, value,
   std::pair<str,int>);
```

When PDG codes + context integers are used with these functions to specify a particle, the **GAMBIT** particle database will be used to translate the code into a string identifier, which will then be checked against the string keys defined in the underlying `SubSpectrum` wrapper to do the data lookup. *When writing new wrappers it is therefore important to ensure that particle string names match the strings registered in the GAMBIT particle database.*

Similarly, whenever any of these interface functions are called with a string name (potentially also with one index), the underlying wrapper will first be searched for a match to that string. If a match cannot be found, then the **GAMBIT** particle database will be used to try to translate the string name into a short name plus an index (e.g. `"~chi_0"` will be translated to the pair (`"~chi"`,0)) or vice versa. If there is still no match found with any spectrum contents, the **GAMBIT** particle database will again be used to try to translate the string name into the string for a matching antiparticle (e.g. `"~chi+_0"` will be translated to `"~chi-_0"`) and the search will be tried again (including translating from string name to short name plus index and vice versa). If no match can be found, an error will be thrown and the scan aborted (except in the case of the `has` functions, which will simply return `false`).

These mechanisms together provide considerable convenience when using the `Spectrum` interface in module functions that do not know precisely how the backend spectrum generator has been connected to a given wrapper. As an example, any of the following calls to a `Spectrum` object containing a `SubSpectrum` that conforms to the `MSSM` spectrum contents (see Appendix C) will retrieve the same information (the lightest selectron/anti-selectron pole mass):

```
spectrum.get(Pole_Mass,"~e-",1);
spectrum.get(Pole_Mass,"~e-_1");
spectrum.get(Pole_Mass,std::make_pair
                        ("~e-",1));
spectrum.get(Pole_Mass,1000011,0);
spectrum.get(Pole_Mass,std::make_pair
                        (1000011,0));

spectrum.get(Pole_Mass,"~e+",1);
spectrum.get(Pole_Mass,"~e+_1");
spectrum.get(Pole_Mass,std::make_pair
                        ("~e+",1));
spectrum.get(Pole_Mass,-1000011,0);
spectrum.get(Pole_Mass,std::make_pair
                        (-1000011,0));
```

There is one final set of overloads to be mentioned: the `safeget` overloads. These act as drop-in replacements for any of the `get` functions, however, they will throw a **GAMBIT**

error (causing the program to halt and report a problem) if the retrieved value is not-a-number (NaN).

2.4 Adding support for new models and/or codes

Adding spectrum generators for models not shipped with GAMBIT is an inevitably technical task, as it involves writing an interface to new code. We have, however, endeavoured to make the procedure as orderly as possible. To spare more casual users the full technical details, we have placed these in Appendix F. Here we give only a brief overview of the procedure, along with a checklist of tasks to be completed and links to the sections in the appendix that explain the corresponding details. Some of the jargon used in the checklist has not been necessary to introduce up until now, but rather than explain it here we instead refer readers to the full details in the Appendix.

The bulk of the work is in writing a `SubSpectrum` wrapper class, which will wrap the new spectrum generator and be usable as we describe in Sect. 2.3. There are also several peripheral tasks to be completed, such as writing a **module function** to properly construct the new `SubSpectrum` and package it into a `Spectrum` object, potentially writing new `SubSpectrumContents` definitions (which enforce consistency between wrappers that ostensibly share a physics model), and making additions to the GAMBIT model hierarchy (discussed in Ref. [11]). These required tasks are:

1. **Choose a `SubSpectrumContents` definition** to which the new wrapper will conform, or write a new one (Appendix F.1).
2. **Write a `SpecTraits` specialisation** to define helper types for the new `SubSpectrum` wrapper (Appendix F.2, but see also the extensions with the `Model` and `Input` typedefs used in Appendices F.3 and F.4).
3. **Write the `SubSpectrum` wrapper class** (Appendix F.2). This step can be further broken down into:

   - **Write the `get`/`set` interface functions** to the external code, which return values conforming to the requirements of the chosen `SubSpectrumContents`, and connect them to the `get`/`set` interface via the `fill_getter_maps` and `fill_setter_maps` functions of the wrapper class. Appendices F.2, F.3 and F.4 cover the various allowed types of interface functions and how to properly connect them to the wrapper interface.
   - **Write the special `get_Model` and `get_Input` functions** in the wrapper (if needed by the chosen type of interface function; see Appendices F.3 and F.4)
   - **Write the RGE interface functions** if RGE-running capabilities are desired (Appendix F.6).

4. **Write the module function(s)** that construct and return a `Spectrum` interface object connected to the new wrapper (Appendix F.7).

To illustrate the process concretely, we provide a full worked example of the writing of a complicated wrapper for a FlexibleSUSY spectrum generator in Appendix G.

2.5 Advanced spectrum usage: vacuum stability

In this section, we present a sophisticated example of how the `Spectrum` object and SpecBit module functions can be used, where we calculate stability of the electroweak vacuum for simple Higgs sectors. These functions are included in SpecBit 1.0.0 and will be used in a follow-up to the study of the scalar-singlet dark matter global fit [18]. To explain what these module functions do and why, we first present a brief review of vacuum stability. We then give the details of the corresponding observable and likelihood functions implemented in GAMBIT.

The stability of the SM electroweak vacuum in the absence of new physics below the Planck scale ($M_{Pl} \approx 1.2 \times 10^{19}$ GeV) has been studied extensively [67–79] (see also references within, and Refs. [80–82]). Although the first bounds on the Higgs mass from vacuum stability were estimated over three decades ago [83,84], the experimentally measured 125 GeV Higgs [85,86] has created increased interest in this topic, as this value results in the existence of a high-energy true vacuum state. If the Higgs field tunnelled from the electroweak vacuum to the global minimum at any point in space, a bubble of low-energy vacuum would form, a process known as *bubble nucleation*. This bubble would propagate outwards at very nearly the speed of light [67], converting all space in its future light cone into the true vacuum state, having catastrophic results.

The lifetime of the SM electroweak vacuum does exceed the lifetime of the Universe. It thus exists in a *meta-stable* state, and the SM is not inconsistent with reality. Yet there is still a non-zero probability of decay, such that a transition to the true vacuum is at some point inevitable.

If new physics does exist below the Planck scale, this may dramatically decrease or increase the lifetime of the false vacuum. For example, as shown in Refs. [73,87,88], using effective couplings suppressed by $1/M_{Pl}$, high-energy physics could indeed reduce the lifetime of the electroweak false vacuum down to a fraction of a second. However, even the most minimal extensions to the SM can have a notable impact on the stability of the electroweak vacuum.

Examples of the most recent calculations of the vacuum stability of the SM are those of Refs. [74,79], using three-loop RGEs and two-loop threshold corrections to the quartic Higgs coupling λ at the weak scale. This is referred to as next-to-next to leading order (NNLO). Before these results

the state of the art was next-to leading order (NLO) using two-loop RGEs and one-loop threshold corrections at the weak scale [89–97]. These vacuum stability studies were, however, performed in a flat background spacetime. Studies have also been done with the SM coupled to gravity in a curved spacetime [98,99], motivated by the possibility that the region of the true minimum (typically $> 10^{10}$ GeV) may be influenced by the gravitational field.

There have been numerous studies of vacuum stability in BSM theories, such as scalar extensions of the Higgs sector [62–64,100–104]. In these scenarios the addition of a scalar particle can have a positive contribution to the running of the quartic Higgs coupling, which for certain values of the new couplings can completely remove the high-energy global minimum [64]. See Ref. [18] for a discussion of vacuum stability in a scalar-singlet-extended Higgs-sector model.

Within GAMBIT we currently use FlexibleSUSY [19] to solve the RGEs up to the Planck scale. For a generic model we are limited to two-loop RGEs and one-loop threshold corrections when using SARAH output. However, higher-order RGEs may be added where these are known, such as in the SM. Therefore even for the scalar-singlet model, where we use generated RGEs, this is comparable to the precision of the most recent studies [64,103]. Additionally, as with the GAMBIT spectrum object itself, the vacuum stability module functions may be used in conjunction with any spectrum generator with RGE-running functionality that can be interfaced to GAMBIT.

The details of the likelihood function and the relevant physics used to classify a parameter point as either stable, meta-stable or unstable are presented in the following section.

2.5.1 Likelihood details

In this section we outline the details of the derivation leading to the likelihood function used in the GAMBIT vacuum stability module functions. The electroweak vacuum obtains a high-energy global minimum when the running quartic Higgs coupling becomes negative at a high scale. For a large renormalisation scale μ the Higgs potential can be approximated with an effective potential of the form

$$V(h) = \frac{\lambda(\mu)}{4} h^4 \tag{1}$$

for $h \gg v$, where $\mu = \mathcal{O}(h)$. The scale at which the quartic coupling becomes negative, and thus the size of the potential barrier between the electroweak vacuum and this global minimum, will affect the likelihood of a quantum mechanical tunnelling to this lower-energy state.

If there exists a high-energy global minimum, then there is a non-zero probability of decay to this state, and thus the electroweak vacuum has a finite lifetime.

There are three possible cases for electroweak vacuum stability, outlined below.

- Stable: If $\lambda(\mu) > 0$ for all $\mu < M_{\text{Pl}}$ then the electroweak vacuum is the only minimum of the Higgs potential (up to M_{Pl}) and is therefore absolutely stable.
- Meta-stable: If there exists a $\mu_0 < M_{\text{Pl}}$ such that $\lambda(\mu_0) < 0$ *and* the lifetime of the electroweak vacuum state is longer than the age of the Universe.
- Unstable: If there exists a $\mu_0 < M_{\text{Pl}}$ such that $\lambda(\mu_0) < 0$ and the lifetime of the electroweak vacuum state is less than the age of the Universe.

Due to the shape of the resulting likelihood function, the distinction between meta-stability and instability is very well defined, with the expected lifetime of the electroweak vacuum being extremely sensitive to changes in the model parameters.

To determine the stability of a model we calculate the rate of quantum mechanical tunnelling through an arbitrary potential barrier, following the derivation in Ref. [105] obtain an analogous field-theoretic solution. In this context, a tunnelling event corresponds to a bubble nucleation in which the Higgs field decays to the lower-energy state. We invoke the thin-wall approximation [106], in which the equation of motion of the bubble of radius R, nucleated at time $t = 0$, is $\left(|\mathbf{x}|^2 - t^2 \right) = R^2$.

From quantum mechanics, one can derive the rate of tunnelling for a particle through a potential barrier, which can then be generalised to a rate of bubble nucleation per unit volume per unit time of

$$\Gamma \approx \Gamma_0 e^{-B/\hbar}, \tag{2}$$

where Γ_0 depends on the size of the past light cone and B is determined by the shape and size of the potential barrier.

Consider a scalar field ϕ. For bubble nucleation in flat four-dimensional spacetime we can take ϕ to be a function of $\rho = \left(\tau^2 + |\mathbf{x}|^2 \right)^{1/2}$ only (where $\tau = it$), as there exists an $O(4)$ symmetry. B is given by the action [105]

$$B = S_E \equiv 2\pi^2 \int_0^\infty \rho^3 \mathrm{d}\rho \left[\frac{1}{2} \left(\frac{\mathrm{d}\phi}{\mathrm{d}\rho} \right)^2 + V \right] \tag{3}$$

satisfying the Euclidean equation of motion for ϕ,

$$\frac{\mathrm{d}^2\phi}{\mathrm{d}\rho^2} + \frac{3}{\rho} \frac{\mathrm{d}\phi}{\mathrm{d}\rho} = \frac{\mathrm{d}V}{\mathrm{d}\phi}. \tag{4}$$

The boundary conditions of this equation of motion can be combined into the single requirement that $\lim_{\rho \to \infty} \phi(\rho) = q_0$. The solution to Eq. (4) is known as the bounce solution, as it describes, in Euclidean time, a solution that is in the unstable vacuum at $t \to -\infty$, the global vacuum at $t = 0$, and bounces back to the starting point at $t \to \infty$.

The process outlined above for a simple scalar field can be applied to the Higgs field, with the use of the approximation $V(h) \approx \lambda(h)h^4/4$. If we perform this calculation at tree level, we can obtain an approximation for the coefficient B. Taking $\lambda(h) = \lambda$, $\lambda < 0$, we have a solution [91] to the analogue of Eq. (4) as

$$h(\rho) = \sqrt{\frac{2}{|\lambda|}} \frac{2R}{\rho^2 + R^2} \tag{5}$$

where R is a dimensional factor associated with the size of the bounce. This can be a relevant quantity such as the height of the barrier, or the change in renormalisation scale between adjacent minima; we shall use the latter. From Eq. (3) we then obtain the action

$$S_E = \frac{8\pi^2}{3|\lambda|}. \tag{6}$$

The validity of the approximation $V \approx \lambda h^4/4$, with $\lambda < 0$, is not immediately evident, as $h = 0$ is an unstable maximum, and indeed any value of $h > 0$ is unstable. However, as stated in Ref. [91], the bounce solution is not a constant field configuration, so requires a non-zero kinetic energy. Because of this, the required bounce solution is suppressed even in the absence of a potential barrier, and thus this is still a valid approximation [107].

Now that we have an expression for B we need to determine the pre-factor. The explicit form of this pre-factor, Γ_0, was first calculated in Ref. [108] by taking account of one-loop quantum corrections. Yet because the behaviour of the exponential in Γ dominates to such a large extent, the value of Γ_0 need only be an approximation, so we follow the analysis of Ref. [67] and determine Γ_0 by a dimensional reasoning.

If we take the Planck constant and the speed of light to be unity ($\hbar = c = 1$), then the rate of decay per unit time per unit volume, Γ/V has units of [length]$^{-4}$. Thus Γ_0 must have units of [length]$^{-4}$ or [energy]^4. The characteristic scale relevant in this problem is the width or height of the potential barrier, thus we take $\Gamma_0 \approx \Lambda_B^4 = 1/R^4$, where we henceforth take Λ_B to be the energy at which $\lambda(\mu)$ is at a minimum.[11]

Γ/V is the rate of decay per unit time per unit volume. As we are ultimately interested in the probability of the Universe having decayed in our past light cone, we multiply Γ/V by

the volume of the past light cone $\sim T_U^4$, where T_U is the age of the Universe.[12] Thus we obtain a rate

$$\Gamma \approx (T_u \Lambda_B)^4 e^{-S_E}, \tag{7}$$

and we obtain the total expected decay rate in the past light cone

$$p \approx \left(e^{140} \frac{\Lambda_B}{M_{\text{Pl}}}\right)^4 \exp\left(-\frac{8\pi^2}{3|\lambda(\Lambda_B)|}\right). \tag{8}$$

Here $\lambda(\Lambda_B)$ is the minimum value of the quartic Higgs coupling, and we have expressed the age of the Universe as $T_u \approx e^{140}/M_{\text{Pl}}$.[13]

The arguments used to arrive at Eq. (8) were based only on dimensional analysis. Although this quantity is now dimensionless it cannot be immediately interpreted as a probability, instead it should be interpreted as the expected value for the random variable k, where k is the number of decay events that occurred in the time given (in this case ≈ 10 billion years).[14] To model the probability that the Universe has actually decayed in the given time interval, we use a Poisson distribution, $\Theta(k; \lambda) = (\lambda^k/k!)e^{-\lambda}$. Because we want the likelihood that no decay has occurred in our past light cone, we calculate the probability that $k = 0$, which is given by

$$\mathcal{L} = \exp\left[-\left(e^{140}\frac{\Lambda_B}{M_{\text{Pl}}}\right)^4 \exp\left(-\frac{8\pi^2}{3|\lambda(\Lambda_B)|}\right)\right]. \tag{9}$$

The likelihood given by Eq. (9) is typically either extremely small or exactly one, being extremely sensitive to the value of $\lambda(\Lambda_B)$. This results in an almost step-function transition from a meta-stable to an unstable Universe when actual Lagrangian-level model parameters are varied.

The likelihood in Eq. (9) is difficult to study, due to its double exponential behaviour. In GAMBIT we return the log-likelihood, so this likelihood contains some useful information even within the stable/meta-stable region of the parameter space. In most cases though, this function will either

---

[11] If the minimum value of the running quartic coupling $\lambda_{\min} < 0$ is achieved after M_{Pl}, but $\lambda(M_{\text{Pl}}) < 0$, we take $\lambda_{\min} = \lambda(M_{\text{Pl}})$.

[12] This can be computed more rigorously for a standard FLRW cosmology, see Ref. [74], however, to the level of detail required here this result is equivalent.

[13] Equation (8) may be expressed in different units. For example, Ref. [67] expresses the age of the Universe in "units of the electroweak scale", as $T_u \approx e^{101}$, where the mass of the Z boson is set as $m_z = 1$. Similarly Ref. [109] expresses the probability in what would equivalently be called "units of the Planck scale", but leave the original factor of M_{Pl} in the expression; this is the style we follow.

[14] By a decay event we mean the decay of the Universe at our position due to a decay at some point in our past light cone, and thus a decay of the observable Universe. Of course, more than one decay event does not physically make sense, so all situations where $k \geq 1$ are effectively equivalent – the Universe has decayed to the true vacuum.

entirely rule out a point or contribute zero to the total log-likelihood. This is indeed useful for ruling out large regions of a parameter space quickly.

2.5.2 Code description

The vacuum stability calculations in GAMBIT are separated into several module functions, shown in Table 3. As renormalisation group running requires a model spectrum object, these include `Spectrum` **dependencies**. The spectrum must be capable of running to different energy scales, and as such cannot be a simple container spectrum.

The module function `find_min_lambda`, which has **capability** `vacuum_stability`, runs the quartic Higgs coupling to determine the scale at which it reaches a turning point. This function produces a `struct` of type `dbl_dbl_bool`, which contains the expected lifetime of the Universe (`double`), the scale of the instability (`double`), and whether or not the couplings remain perturbative up to the scale returned (`bool`). If a turning point is not reached before some user-specified high scale (set via YAML option `set_high_scale`, which defaults to M_{Pl}), the high scale is returned instead of the scale of the turning point. These calculations can be applied to any model with a Higgs field that does not mix with other scalars.

Derived from the results of `find_min_lambda` is the physical observable

– `expected_lifetime`: the expected lifetime of the electroweak vacuum in years

and the likelihood functions

– `VS_likelihood`: the log-likelihood of vacuum decay not having yet occurred, given the current age of the Universe
– `check_perturb_min_lambda`: the perturbativity of the running couplings at the instability scale. To allow this function to be used as a log-likelihood, it returns 0 for models where the couplings remain perturbative up to the instability scale, and invalidates the model otherwise.

In `VS_likelihood`, the expected age is compared to the current age of the Universe, which is currently hard-coded to 13.6×10^9 years. If λ does not have a turning point before M_{Pl}, then the lifetime returned is 1×10^{300} years.

The function `check_perturb_min_lambda` is an efficient way to check that the couplings of a theory remain perturbative up to M_{Pl} or some alternative scale, chosen via the option `set_high_scale` of function `find_min_lambda`. By performing this check directly in GAMBIT, we avoid any need to rely on similar checks implemented in some (but not all) spectrum generators. The actual calculation is performed simultaneously with the vacuum stability check, and is thus

extracted by `check_perturb_min_lambda` from the output of the `vacuum_stability` capability. The result is a flag set `true` only if all spectrum parameters tagged `dimensionless` are less than $(4\pi)^{1/2}$ at a set of points evenly distributed in log space between the electroweak scale and the smaller of Λ_B and the high scale chosen for `find_min_lambda`. The default is to test 10 points; this can be reset with the YAML option `check_perturb_pts` of `find_min_lambda`.

A simple example of the GAMBIT input YAML commands for using the vacuum stability likelihood, and outputting the Higgs pole mass and the expected age of the Universe as observables, is given below.

```
  - capability: print_SingletDM_spectrum
    purpose: Observable

  - capability: VS_likelihood
    purpose: LogLike

  - capability: expected_lifetime
    purpose: Observable
```

2.6 Higgs couplings

In addition to providing particle spectra to the rest of GAMBIT, SpecBit can also produce Higgs couplings, which it provides as a compact `HiggsCouplingsTable`. The Higgs couplings are required by HiggsBounds/HiggsSignals, which are discussed in the ColliderBit manual, Ref. [14]. These objects carry all decays of neutral and charged Higgs bosons present in the theory under investigation, their CP eigenvalues, a list of all particles in the theory that Higgses can decay invisibly to, values of the SM-normalised effective couplings of each neutral Higgs to WW, ZZ, $t\bar{t}$, $b\bar{b}$, $c\bar{c}$, $s\bar{s}$, $\tau^+\tau^-$, $\mu^+\mu^-$, $Z\gamma$ and $h\gamma$, and additional comparison decays of SM Higgses with the same masses as the neutral BSM Higgses. SpecBit obtains all decays from DecayBit. It makes the CP and invisible state identifications within its own functions. The effective couplings can be obtained in two different ways. The first is to build them directly from couplings provided by dedicated routines in external calculators, as in `MSSM_higgs_couplings_FH` (see Table 2), which retrieves couplings from FeynHiggs (see Sect. 3.1.3 for details of this interface). Alternatively, they can be estimated on the basis of the partial width approximation, where the squared effective coupling is taken as the ratio of the partial widths to the relevant final state of the BSM and equivalent SM-like Higgs. Examples of the partial width approach are `MSSM_higgs_couplings_pwid` (Table 2) and `SingletDM_higgs_couplings_pwid` (Table 3).

Note that all Higgs couplings available in GAMBIT presently refer to loop-corrected values; tree-level couplings could be added if required, but should be differentiated from

Table 3 **Capabilities** provided by SpecBit for the SingletDM model. The **capability** `SingletDM_spectrum` is understood to provide running parameters in the $\overline{MS}$ scheme, along with pole masses, with low-energy Standard Model inputs also defined in the $\overline{MS}$ scheme

| Capability	Function (return type): brief description	Dependencies	Options (type)
`SingletDM_spectrum`	`get_SingletDM_spectrum_simple` (`Spectrum`): Create simple container `Spectrum` object from `SMInputs` structures, SM Higgs parameters, and the SingletDM parameters	`SMINPUTS`	
	`get_SingletDM_spectrum_FS` (`Spectrum`): Create `Spectrum` object with RGE facilities for the scalar-singlet model using FlexibleSUSY	`SMINPUTS`	FlexibleSUSY
	`get_SingletDM_spectrum_as_map` (`map_str_dbl`): Convert a SingletDM spectrum into a `std::map` for printing	`SingletDM_spectrum`	

`Higgs_Couplings`	`SingletDM_higgs_couplings_pwid` (`HiggsCouplingsTable`): Construct a table of Higgs couplings from Higgs partial decay widths	`SingletDM_spectrum`
		`Reference_SM_Higgs_decay_rates`
		`Higgs_decay_rates`	

`vacuum_stability`	`find_min_lambda` (`triplet<double>`): Run the quartic Higgs coupling to high scales, returning a `triplet` containing the expected lifetime of the electroweak vacuum, the instability scale, and the perturbativity of the couplings up to the requested scale	`SingletDM_spectrum`	`set_high_scale` (`double`) `check_perturb_pts` (`int`)
`check_perturb_min_lambda`	`get_check_perturb_min_lambda` (`double`): Check that dimensionless couplings are sufficiently small up to the smaller of the Planck scale and the scale at which the quartic Higgs coupling is minimised *and* non-positive	`vacuum_stability`	
`VS_likelihood`	`get_likelihood` (`double`): Get the *a priori* likelihood of no electroweak vacuum decays having occurred in the past light cone	`vacuum_stability`	
`expected_lifetime`	`get_expected_lifetime` (`double`): Return the expected lifetime of the electroweak vacuum in units of years	`vacuum_stability`	
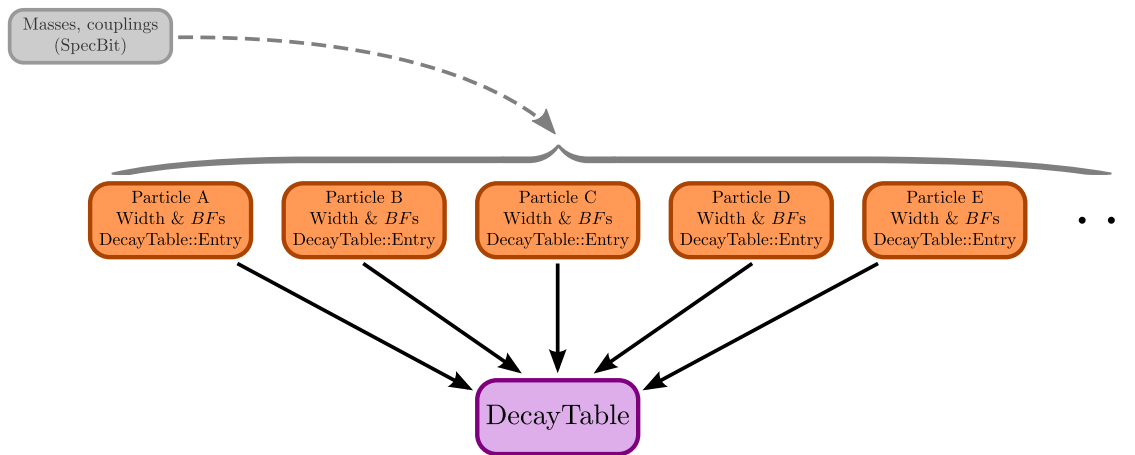
Fig. 8 Schematic representation of the structure of DecayBit. Masses and couplings are provided from SpecBit (grey box), and used to calculate decay widths and branching fractions (BFs) for each particle individually, sometimes with the assistance of backends such as SUSY-HIT and FeynHiggs. The widths and BFs are gathered into a single DecayTable::Entry for each particle (orange boxes). These are then collected into a single GAMBIT DecayTable (purple box), containing all decays of the entire spectrum, and eventually passed on to other GAMBIT modules

the present couplings by assigning them an alternative capability (e.g. `Higgs_Couplings_tree_level`).

3 DecayBit

All particle decay widths used in GAMBIT are provided by the dedicated module DecayBit. These range from known decays of SM particles, to modifications of SM decay widths and branching fractions due to new physics (e.g. top and Higgs decays), to decay widths of new particles present in BSM theories.

These data are needed by ColliderBit [14] for predicting Higgs decay rates and calculating corresponding LHC Higgs likelihoods. They also allow ColliderBit to connect predictions for LHC production of BSM particles with observable final states, by simulating particle collisions and subsequent decays of the particles produced in the initial interaction. The same data are required by DarkBit [13] to simulate cascade decays of final states produced in dark matter annihilation and decay, allowing it to make predictions for cosmic ray fluxes in dark matter indirect detection experiments. Decay widths of SM and BSM particles are also important for predicting the relic density of dark matter, through their participation in thermal freeze-out, or if dark matter is produced non-thermally from decays of some heavy resonance.

Although DecayBit provides uncertainties on some decay widths and branching fractions (currently only those that we take from the PDG [61]), these are not utilised as yet by other GAMBIT modules (indeed not even by HiggsSignals nor HiggsBounds). Future likelihood functions are expected to make use of these data, and future versions of DecayBit

are anticipated to also feature calculation of uncertainties on partial and total widths of BSM states.

The overall structure of DecayBit is outlined in Fig. 8, showing the modular calculation of each particle's decays, and their combination into a complete DecayTable for the model under investigation. Here we begin by describing the decays currently implemented in DecayBit (Sect. 3.1), including the details of the calculations, input data and external codes it employs. We then go through the specific module functions it offers (Sect. 3.2), and the details of the code (Sect. 3.3), covering the associated data structures and utility functions, how to add support for new channels and models, and an explicit worked example of the computation of Higgs boson decays.

3.1 Supported decays

3.1.1 Standard model

DecayBit contains total widths and associated uncertainties for decays of the W, Z, t, b, τ and μ, as well as their antiparticles, and the most common mesons π^0, $\pi^\pm$, η, ρ^0, $\rho^\pm$ and ω. It also includes partial widths to all observed, distinct final states for W, Z, t, b, τ, μ, π^0 and $\pi^\pm$, including many-body final states in many cases (although we do not include such states where the branching fraction would otherwise be contained within a lower-multiplicity channel). We take these data directly from the Particle Data Group compilation of experimental results [61]. These data are used in GAMBIT in two scenarios. The first is whenever a decay has no relevant BSM contribution, e.g. if BSM contributions are heavily loop-suppressed, or if the particle sector of the scan is sim-

ply the SM, as when scanning e.g. the DM halo model only. The second is in the cases where the only significant BSM modification to the decay of the SM particle is to open a new decay channel, meaning that the new decay can simply be added to the existing SM decays. Where neither of these scenarios apply, i.e. where BSM corrections are relevant for SM → SM decays, DecayBit accounts for the corrections via an appropriately model-dependent calculation.

We group together hadronic channels for W and Z decays, assigning them a final state consisting of two generic hadrons. Note that 'hadron' is recognised as a generic particle by the GAMBIT particle database; see Sect. 10.1 of Ref. [11].

DecayBit also features various calculations of Higgs boson partial and total decay widths. For a pure SM Higgs, the user can choose to either calculate decays at the predicted value of the Higgs mass with FeynHiggs, or extract them from precomputed tables contained in DecayBit. The latter are the predicted SM Higgs partial and total widths given as a function of Higgs mass in Ref. [110]. Using these tables wherever possible is preferable to simply calling HDECAY, as they include additional higher-order corrections for 4-body final states obtained using PROPHECY4F [111,112].

3.1.2 Scalar singlet

When $m_S < m_h/2$, the Higgs can decay to two singlet scalars. The partial width for this process is

$$\Gamma_{h \to ss} = \frac{\lambda_{hS}^2 v_0^2}{32\pi \, m_h} \left(1 - 4m_S^2/m_h^2\right)^{1/2}. \tag{10}$$

Whenever this channel is kinematically open, DecayBit calculates the partial width and adds it to the existing set of SM Higgs decays, rescaling the decay branching fractions (BFs) and total width obtained for a pure SM Higgs.

The decay $h \to SS$ would be entirely invisible at the LHC, so would contribute to the invisible width of the Higgs boson. Assuming SM-like couplings, as is the case for the scalar-singlet model, 95% confidence level (CL) upper limits on the Higgs invisible width from LHC and Tevatron data are presently at the level of 19% [113]. We implement this as a complete likelihood function in DecayBit, by interpolating the function shown in Fig. 8 of [113].

3.1.3 MSSM

DecayBit will calculate decays of all sparticles and additional Higgs bosons in the MSSM, including branching fractions to SM and SUSY final states. It can also calculate SUSY corrections to decays of the top quark and the SM-like Higgs.

The decays of each Higgs may come (independently) from backend functions in either HDECAY via SUSY-HIT, or FeynHiggs. For the calculation of effective Higgs couplings

by SpecBit (see Sect. 2.6), DecayBit will also compute decays of an SM Higgs with the same mass as any of the MSSM Higgs bosons, using either FeynHiggs or by interpolating its own internal tables. For masses between 80 GeV and 1 TeV, these tables come directly from Ref. [110]. Between 1 and 80 GeV, and between 1 and 16 TeV, we supplement them with additional results that we obtained by running HDECAY 6.51 [28]. For Higgs masses above 1 TeV, in order to obtain finite results we disabled electroweak corrections and decays to $s\bar{s}$, $b\bar{b}$, gg and $\mu^+\mu^-$ (which are negligible at large masses, where W^+W^-, ZZ and $t\bar{t}$ final states dominate). Because of these settings, and the absence of any corrections obtained with PROPHECY4F [111,112] in our supplemental calculations, small discontinuities can be expected at the transitions between different tables.

Top quark decays are available only by calling FeynHiggs. Sparticle decays are available only from SDECAY via SUSY-HIT. DecayBit can calculate Higgs and top widths for arbitrary MSSM-63 models when employing FeynHiggs, but is limited to MSSM-20 models for sparticle and Higgs decays when using SUSY-HIT, due to the fact that SUSY-HIT is not SLHA2 compliant as models with greater freedom rely on the more general SLHA2 format allowing full family mixing amongst the sfermions.

We backend SUSY-HIT 1.5 in a similar fashion to other codes, but we patch it via the GAMBIT build system when it is automatically downloaded, in order to make a number of crucial modifications. Here we go through the most important of these.

SUSY-HIT is written as a standalone program, whereas GAMBIT requires callable functions in a shared library. We therefore convert the main program of SUSY-HIT to a function, and modify the makefile to produce a shared library.

Interfacing to HDECAY and SDECAY via SUSY-HIT means that we retain the standard HDECAY and SDECAY options chosen there: multi-body decays, loop-induced decays and 1-loop QCD corrections to 2-body decays are all enabled, and any running masses and couplings are calculated at the EWSB scale in the $\overline{DR}$ scheme. However, we directly inject the bottom quark pole mass calculated by SpecBit into SUSY-HIT, rather than relying on its internal pole mass calculation.

For some rather specific models, the 1-loop QCD corrections Δ_1 to the widths of the lighter sparticles are negative and larger than the tree-level results, causing SUSY-HIT to return negative decay widths. We have so far found models where this is the case for gluinos, sbottoms, stops, sups, sdowns, neutralinos and charginos. To rectify this, we have patched SUSY-HIT to implement a correction factor designed to approximately mimic resummation. When the corrections are negative and larger than the tree-level result, instead of obtaining the 1-loop corrected widths as usual in SUSY-HIT

$$\Gamma_{\text{tot}} = \Gamma_{\text{tree}}(1 + \Delta_1), \tag{11}$$

we apply the correction as

$$\Gamma_{\text{tot}} = \frac{\Gamma_{\text{tree}}}{1 - \Delta_1}, \tag{12}$$

motivated by a similar improvement implemented in NMS-DECAY [114]. Here $\Delta_1 = \Delta_{X,1} + \Delta_{Y,1} + \dots$ refers to the sum of 1-loop corrections to the partial widths for all 2-body final states X, Y, etc. Note that of course this expression does not *actually* resum higher-order corrections correctly. It is simply employed as an expression that reproduces the leading-order term of Eq. (11) and will be positive, so that the pathology of the negative width is avoided.

With the original definition (Eq. (11)) it is thus straightforward to define a 1-loop 2-body branching fraction $\mathcal{B}$ for final state X as

$$\mathcal{B}_X = \Gamma_{X,\text{tree}}(1 + \Delta_{X,1})/\Gamma_{\text{tot}}. \tag{13}$$

However, this no longer follows when Γ_{tot} is corrected by Eq. (12). In this case, we simply revert to determining branching fractions from the tree-level partial and total widths, but retain the more accurate total width obtained by Eq. (12) as the total decay width to be used elsewhere in GAMBIT.

Correcting negative widths with Eq. (12) has a downside: the total width of a sparticle becomes discontinuous in the model parameters at $\Delta_1 = -1$. Whilst this is still an obvious improvement over having non-physical results for $\Delta_1 < -1$, it is not ideal. An alternative treatment would be to use Eq. (12) whenever $\Delta_1 < 0$, which would ensure continuity, and to apply the correction to each partial width individually, allowing one-loop branching fractions to still be calculated when $\Delta_1 < 0$. However, this would modify the standard behaviour of SUSY-HIT in the regime $-1 < \Delta_1 < 0$. In the interests of keeping our modifications to external codes simple and minimal, we therefore consider it preferable to pay the price of a discontinuous width, restricting our modifications to changes that only have an impact when SUSY-HIT would otherwise return unphysical results.

A related issue is that at high SUSY-breaking scales, the $\overline{MS}$ correction to the tree-level t and b pole masses required for calculating Higgs couplings in SUSY-HIT also becomes large, leading to numerical instabilities. To remedy this, we import the improved running $\overline{MS}$ expressions for m_t and m_b from HDECAY 6.51 to SUSY-HIT.

Finally, we also implemented a diskless generalisation of the SLHA interface to SUSY-HIT. This avoids the need to read and write files to disk, allowing GAMBIT spectrum contents to be directly injected into SUSY-HIT common blocks in SLHA format, and decay data to be extracted directly

from other common blocks and easily converted to the native GAMBIT `DecayTable` format (described in Sect. 3.3).

We connect to and use FeynHiggs as a regular GAMBIT backend, without significant[15] modification. GAMBIT supports versions 2.11.2 and later. We adopt the settings for the real MSSM recommended in the FeynHiggs documentation, but restrict neutral Higgs mixing to the CP-even states, for compatibility with HiggsBounds/HiggsSignals. For version 2.12.0 we set `loglevel = 3` in order to include NNLL corrections.[16] We pass MSSM models to FeynHiggs using the function `FHSetPara`, directly setting the SM parameters in SLHA2 parametrisation, the CP-odd Higgs pole mass, the μ parameter at the SUSY scale and the $\overline{DR}$ soft terms (sfermion mass parameter matrices, gaugino mass parameters and trilinear couplings, also at the SUSY scale), from the outputs of the `Spectrum` objects provided by SpecBit. We set the `FHSetPara` arguments `Qt` and `Qb` to the renormalisation scale of the $\overline{DR}$ soft masses, so that FeynHiggs interprets stop- and sbottom-sector parameters in the $\overline{DR}$ scheme, and converts them internally to the on-shell scheme.

3.2 Available functions and options

Here we give a compact census of the module functions in DecayBit. These are also laid out in Tables 4, 5, 6 and 7.

Antiparticle equivalents of all functions described below are automatically created from the decays of their particle partners, assuming CP symmetry. This need not be assumed when adding additional states or channels to DecayBit.

3.2.1 Standard model

The SM particle decay functions are listed in Table 4. These are assigned names

  particle_designation`_decays`

and **capabilities**

  particle_designation`_decay_rates`,

where *particle_designation* is simply the particle name for neutral isosinglets, *particle_designation* = *particle_name_*`{plus, minus}` for charged leptons and gauge bosons, *particle_designation* = {*particle_name*, *particle_name*bar} for quarks, and *particle_designation* = *particle_name_*`{0, plus, minus}` for isotriplet mesons.

---

[15] We correct some minor bugs preventing compiler safety checks.

[16] We make no attempt within GAMBIT to directly remedy any discrepancies seen between FeynHiggs and other codes [55,66]; we refer the reader to those papers, and the first GAMBIT MSSM results papers [16,17] for further discussion.

Table 4 SM decay functions provided by DecayBit. None of these functions has any backend requirements. Here X is either empty or equal to bar, $Y \in$ {plus, minus} and $Z \in$ {0, plus, minus}. All functions with X=bar, Y=minus or Z=minus have a dependency on their CP-conjugate equivalent, with X empty, Y=plus or Z=plus

Capability	Function (return type): brief description	Dependencies	Options (type)
tX_decay_rates	tX_decays (DecayTable::Entry): Computes all SM decays of the $t/\bar{t}$ quark		
mu_Y_decay_rates	mu_Y_decays (DecayTable::Entry): Computes all SM decays of the μ^+/μ^- lepton		
tau_Y_decay_rates	tau_Y_decays (DecayTable::Entry): Computes all SM decays of the τ^+/τ^- lepton		
pi_Z_decay_rates	pi_Z_decays (DecayTable::Entry): Computes all SM decays of the π^0, π^+ and π^- mesons		
rho_Z_decay_rates	rho_Z_decays (DecayTable::Entry): Computes all SM decays of the ρ^0, ρ^+ and ρ^- mesons		
eta_decay_rates	eta_decays (DecayTable::Entry): Computes all SM decays of the η meson		
omega_decay_rates	omega_decays (DecayTable::Entry): Computes all SM decays of the ω meson		
Z_decay_rates	Z_decays (DecayTable::Entry): Computes all SM decays of the Z boson		
W_Y_decay_rates	W_Y_decays (DecayTable::Entry): Computes all SM decays of the W^+/W^- gauge boson		
Reference_SM_Higgs_decay_rates	Ref_SM_Higgs_decays_table (DecayTable::Entry): Computes all decays of the Higgs boson in the SM, by interpolating in the tables of Ref. [110]	mh	higgs_minmass (double)
	Ref_SM_Higgs_decays_FH (DecayTable::Entry): Computes all decays of the Higgs boson in the SM, using Higgs couplings obtained from FeynHiggs	SMlike_Higgs_PDG_code	higgs_maxmass (double)
Higgs_decay_rates	SM_Higgs_decays (DecayTable::Entry): Returns all decays of the SM Higgs boson	FH_Couplings_output SLHA_pseudonyms Reference_SM_Higgs_decay_rates	

Table 5 Scalar-singlet functions provided by DecayBit. Neither of these functions has any backend requirements nor options

Capability	Function (return type): brief description	Dependencies
`Higgs_decay_rates`	`SingletDM_Higgs_decays` (`DecayTable::Entry`): Computes all decays of the Higgs boson, including to scalar singlets	`SingletDM_spectrum` `Reference_SM_Higgs_decay_rates`
`lnL_Higgs_invWidth`	`lnL_Higgs_invWidth_SMlike`(`double`): Computes the log-likelihood of the Higgs invisible width	`Higgs_decay_rates`

The function

`Ref_SM_Higgs_decays_table`

constructs `Reference_SM_Higgs_decay_rates` from the tables of Ref. [110] (which are determined using HDECAY [28–30] and PROPHECY4F [111,112]), using the pole mass provided by SpecBit or PrecisionBit (see Sect. 3.1.1). The alternative function

`Ref_SM_Higgs_decays_FH`

constructs `Reference_SM_Higgs_decay_rates` from Higgs couplings obtained from FeynHiggs (by e.g. `SpecBit::FH_Couplings`; see Table 2). The result of one or the other of these two functions is presented to the rest of GAMBIT as the definitive `Higgs_decay_rates`, by function

`SM_Higgs_decays`.

The two functions are also used by other module functions to compare BSM Higgs decays to equivalent SM decays.

3.2.2 Scalar singlet

We highlight the parts of DecayBit specific to the scalar-singlet model in Table 5. Only the `Higgs_decay_rates` are modified in this model. The function

`SingletDM_Higgs_decays`

simply piggybacks off the SM Higgs decay calculation, and then rescales the total width and branching fractions to accommodate the additional $h \rightarrow SS$ contribution. The SM-like Higgs invisible width likelihood is provided by the function

`lnL_Higgs_invWidth_SMlike`,

which has **capability** `lnL_Higgs_invWidth` and depends on the `Higgs_decay_rates`.

3.2.3 MSSM

Table 6 lists the full set of MSSM decay functions contained in DecayBit. The naming scheme follows a similar structure of the SM decays, with function names

particle_designation`_decays`

and **capabilities**

particle_designation`_decay_rates`.

Here *particle_designation* includes the particle name and any relevant indices: `plus` or `minus` for charged bosons, `bar` for sfermionic antiparticles, mass-ordering indices for bosons, neutralinos and charginos and third-generation sfermions, and weak eigenstate indicators `l` or `r` for first- and second-generation sfermions. Despite the fact that GAMBIT works exclusively in SLHA2 mass-ordered basis internally, we designate first- and second-generation sfermion decays in the weak eigenstate basis because SUSY-HIT is not SLHA2-compliant, and left–right mixing is typically small in the first two generations in the MSSM anyway. This will probably be generalised somewhat further in future revisions of DecayBit. We discuss the conversion between mass, weak and family-mass eigenstates in more detail in Sect. 3.3.2.

DecayBit can compute MSSM Higgs sector decays using either SUSY-HIT or FeynHiggs; it includes separate module functions for each of these options.[17] It also includes additional top decays via intermediate off-shell MSSM states, calculable via FeynHiggs. Similar to the reference functions for the SM-like Higgs described in Sect. 3.2.1, DecayBit also provides functions to compute reference decays of an SM Higgs with the same mass as the A^0 (capability `Reference_SM_A0_decay_rates`) and the 'other', non-SM-like, CP-even neutral MSSM

---

[17] Note that the only module functions in Table 6 with an explicit backend requirement listed are those that make use of SUSY-HIT. The corresponding FeynHiggs functions instead depend on `FH_Couplings_output`, which can be obtained from SpecBit's function `FH_Couplings` – which exhibits the equivalent backend requirement on FeynHiggs (see Table 2).

Table 6 MSSM decay functions provided by DecayBit. None have any options of note. Here A and B are mass indices that run from 1 to 2 and 1 to 4, respectively, C is a helicity label (`l` or `r`), and X and Y are as in Table 4 (i.e. empty/`bar` and `plus`/`minus`)

Capability	Function (return type): brief description	Dependencies	Backend requirements
`t_decay_rates`	`FH_t_decays(DecayTable::Entry)`: Computes MSSM decays of the t quark	`FH_Couplings_output`	
`Higgs_decay_rates`	`MSSM_h0_1_decays(DecayTable::Entry)`: MSSM decays of the lightest CP-even Higgs with SUSY-HIT	`SLHA_pseudonyms`	SUSY-HIT
	`FH_MSSM_h0_1_decays(DecayTable::Entry)`: MSSM decays of the lightest CP-even Higgs with FeynHiggs	`SLHA_pseudonyms` `FH_Couplings_output`	
`h0_2_decay_rates`	`h0_2_decays(DecayTable::Entry)`: Decays of the heaviest CP-even MSSM Higgs with SUSY-HIT	`SLHA_pseudonyms`	SUSY-HIT
	`FH_h0_2_decays(DecayTable::Entry)`: Decays of the heaviest CP-even MSSM Higgs with FeynHiggs	`SLHA_pseudonyms` `FH_Couplings_output`	
`A0_decay_rates`	`A0_decays(DecayTable::Entry)`: Decays of the CP-odd MSSM Higgs boson with SUSY-HIT	`SLHA_pseudonyms`	SUSY-HIT
	`FH_A0_decays(DecayTable::Entry)`: Decays of the CP-odd MSSM Higgs boson with FeynHiggs	`SLHA_pseudonyms` `FH_Couplings_output`	
`HY_decay_rates`	`HY_decays`: Decays of the MSSM charged Higgs with SUSY-HIT	`SLHA_pseudonyms`	SUSY-HIT
	`FH_HY_decays`: Decays of the MSSM charged Higgs with FeynHiggs	`SLHA_pseudonyms` `FH_Couplings_output`	

Table 6 continued

Capability	Function (return type): brief description	Dependencies	Backend requirements
Reference_SM_other_Higgs_decay_rates	Ref_SM_other_Higgs_decays_table (DecayTable::Entry): Computes all decays of an SM Higgs with mass equal to the mass of the *least* SM-like CP-even Higgs boson in the MSSM, by interpolating in the tables of Ref. [110] and extensions computed with HDECAY 6.51	SMlike_Higgs_PDG_code MSSM_spectrum	
	Ref_SM_other_Higgs_decays_FH (DecayTable::Entry): Computes all decays of an SM Higgs with mass equal to the mass of the *least* SM-like CP-even Higgs boson in the MSSM, using Higgs couplings obtained from FeynHiggs	SMlike_Higgs_PDG_code FH_Couplings_output SLHA_pseudonyms	
Reference_SM_A0_decay_rates	Ref_SM_A0_decays_table (DecayTable::Entry): Computes all decays of the CP-odd Higgs boson in the MSSM, by interpolating in the tables of Ref. [110] and extensions computed with HDECAY 6.51	SMlike_Higgs_PDG_code MSSM_spectrum	
	Ref_SM_A0_decays_FH (DecayTable::Entry): Computes all decays of the CP-odd Higgs boson in the MSSM, using Higgs couplings obtained from FeynHiggs	SMlike_Higgs_PDG_code FH_Couplings_output SLHA_pseudonyms	
gluino_decay_rates	gluino_decays (DecayTable::Entry): Computes decays of the gluino	SLHA_pseudonyms	SUSY-HIT
chargino$Y$_A_decay_rates	chargino$Y$_A_decays(DecayTable::Entry): Computes decays of the charginos	SLHA_pseudonyms	SUSY-HIT

Table 6 continued

Capability	Function (return type): brief description	Dependencies	Backend requirements
neutralino_B_decay_rates	neutralino_B _decays(DecayTable::Entry): Computes decays of the neutralinos	SLHA_pseudonyms	SUSY-HIT
supX_C_decay_rates	supX_C _decays(DecayTable::Entry): Computes decays of the $\tilde{u}$ squarks		SUSY-HIT
sdownX_C_decay_rates	sdownX_C _decays(DecayTable::Entry): Computes decays of the $\tilde{d}$ squarks		SUSY-HIT
scharmX_C_decay_rates	scharmX_C _decays(DecayTable::Entry): Computes decays of the $\tilde{c}$ squarks		SUSY-HIT
sstrangeX_C_decay_rates	sstrangeX_C _decays(DecayTable::Entry): Computes decays of the $\tilde{s}$ squarks		SUSY-HIT
stopX_A_decay_rates	stopX_A _decays(DecayTable::Entry): Computes decays of the $\tilde{t}$ squarks	SLHA_pseudonyms	SUSY-HIT
sbottomX_A_decay_rates	sbottomX_ A_decays(DecayTable::Entry): Computes decays of the $\tilde{b}$ squarks	SLHA_pseudonyms	SUSY-HIT
selectronX_C_decay_rates	selectronX_ C_decays(DecayTable::Entry): Computes decays of the $\tilde{e}$ sleptons		SUSY-HIT
smuonX_C_decay_rates	smuonX_ C_decays(DecayTable::Entry): Computes decays of the $\tilde{\mu}$ sleptons	SLHA_pseudonyms	SUSY-HIT
stauX_A_decay_rates	stauX_ A_decays(DecayTable::Entry): Computes decays of the $\tilde{\tau}$ sleptons		SUSY-HIT
snuX_electron1_decay_rates	snuX _electron1_decays(DecayTable::Entry): Computes decays of the $\tilde{\nu}_{e_L}$ sneutrinos		SUSY-HIT
snuX_muon1_decay_rates	snuX _muon1_decays(DecayTable::Entry): Computes decays of the $\tilde{\nu}_{\mu_L}$ sneutrinos		SUSY-HIT
snuX_tau1_decay_rates	snuX _tau1_decays(DecayTable::Entry): Computes decays of the $\tilde{\nu}_{\tau_L}$ sneutrinos	SLHA_pseudonyms	SUSY-HIT

Table 7 Collector and helper module functions provided by DecayBit. Here X refers to all particles that exist in a given model. Note that `str` is a GAMBIT-specific `typedef` of `std::string`

Capability	Function (return type): brief description	Dependencies	Options (type)
decay_rates	all_decays(DecayTable): Computes all decays of all particles in a given model all_decays_from_SLHA(DecayTable): Reads decays from SLHA files	X_decay_rates	drop_SLHA_file(bool) SLHA_output_filename(str) SLHA_decay_filenames(std::vector<str>)
SLHA1_violation	check_first_sec_gen_mixing(int): Checks if first- and second-generation sfermions mix at more than a given %	MSSM_spectrum	gauge_mixing_tolerance(double)
SLHA_pseudonyms	get_mass_es_pseudonyms(DecayBit::mass_es_pseudonyms): Computes nearest-match SLHA2 mass eigenstates for each SLHA1 sfermion weak/family eigenstate	MSSM_spectrum	gauge_mixing_tolerance(double) gauge_mixing_tolerance_invalidates_point_only(bool)

Higgs (usually – but not always – the heavy Higgs; capability `Reference_SM_other_Higgs_decay_rates`). These are necessary for constructing effective Higgs couplings from partial widths with SpecBit (see Sect. 2.6).

3.2.4 Collectors and helpers

DecayBit also ships with a number of other useful module functions, given in Table 7. The most important are the collector functions

> `all_decays` and
> `all_decays_from_SLHA`.

These both have **capability** `decay_rates` and return a complete GAMBIT `DecayTable` object (see Sect. 3.3.1 below). This is a single object with all decays of all known particles in the models under investigation, to be used by the rest of GAMBIT in subsequent physics calculations. The first of these functions constructs the `DecayTable` from the results of the relevant individual decay module functions in DecayBit. The second is a convenience/debug tool that simply bypasses the rest of DecayBit, reading some precomputed decays from a user-supplied SLHA file and presenting them to the rest of the code as if they had been computed by DecayBit.

The other two module functions have to do with the conversion between sfermion weak/family and mass eigenstates. The function

> `check_first_sec_gen_mixing`

checks whether first- or second-generation weak eigenstates are mixed beyond the level specified by the option `gauge_mixing_tolerance`, returning zero if all states are sufficiently pure and an integer between 1 and 6 otherwise, depending on which eigenstate shows the mixing. This function is mostly intended for saving information about mixing for later inspection. The actual sparticle decay functions all have **dependencies** on the `SLHA_pseudonyms`, which list the sparticle mass eigenstates containing the largest admixture of each weak eigenstate; the function that returns this list is

> `get_mass_es_pseudonyms`.

The underlying weak/family–mass eigenstate conversion routines are discussed in more detail in Sect. 3.3.2.

3.3 Code description and interface details

3.3.1 The `DecayTable`

GAMBIT includes a specific `DecayTable` container class for storing and passing decay information to the rest of the code. A `DecayTable` contains one or more instances of the subclass `DecayTable::Entry`, each of which holds the full decay information for a single particle. The primary job of DecayBit is to construct a `DecayTable::Entry` for each particle in a given model, and then to put them together into a complete `DecayTable` that other parts of the code can access.

Each `DecayTable::Entry` includes the width of the particle, the positive and negative uncertainties on that width, the branching fractions to different decay channels, the uncertainties on the branching fractions, and information about which code(s) (GAMBIT, a backend or some combination) was responsible for computing the decay information. The width and branching fraction uncertainties are not currently used in any calculations in GAMBIT, and they are only set for SM particles (where they are known experimentally from the Particle Data Group [61]), but they may be set more completely and used in more advanced likelihood functions in the future, for example in Higgs physics. Widths can be set and accessed directly with the `width_in_GeV` field. Decay channels and branching fractions are added and retrieved with `set_BF()` and `BF()`. The following simple example for the decays of the W^+ boson shows these in action:

```
// Declare a set of decays for the
// W+ DecayTable::Entry Wplus;

// Set the total width
Wplus.width_in_GeV = 2.085;

// Set branching fraction and uncertainty
// for W → e νₑ
Wplus.set_BF(0.1071, 0.0016, "e+", "nu_e");

// Find the partial width for W → e νₑ
double pw =  Wplus.width_in_GeV
            * Wplus.BF("e+","nu_e");
```

The functions `set_BF()` and `BF()` each have five different overloads, allowing the decay channel to be specified with any of the particle identifiers recognised by the GAMBIT particle database (see Sect. 10.1 in Ref. [11]). These are:

1. a simple argument list of strings corresponding to the final state particles' long names (as in the example above)
2. an argument list of alternating PDG codes and context integers,
3. an argument list of alternating short name strings and integer particle indices,
4. a `std::vector` of long name strings, and

5. a `std::vector<std::pair<int,int>>` of PDG code – context integer pairs.

All of these forms allow an arbitrary number of final state particles to be specified for a given decay channel. The `DecayTable::Entry` subclass also has a simple convenience method `has_channel()` that takes channels specified in any of the above forms, and returns a `bool` indicating whether the `Entry` already contains a field corresponding to that decay channel or not.

The individual entries in a full `DecayTable` can be accessed using semantics reminiscent of a regular C++ `std::map`, where the key is the decaying particle, referred to by any of the three possible designators recognised by the GAMBIT particle database. For example,

```
// Declare a DecayTable
DecayTable BosonDecays

// Add entries (here are some I prepared
   earlier)
BosonDecays("W+")    = Wplus;
BosonDecays("W-")    = Wminus;
BosonDecays("Z0")    = Z;
BosonDecays("h0_1") = SM_higgs;

// Get the branching fraction for W → e νₑ
double enueBF =
    BosonDecays("W+").BF("e+","nu_e");

// Get underlying Entry for Higgs
// six different ways
DecayTable::Entry h;
h = BosonDecays("h0_1");
h = BosonDecays(25, 0);
h = BosonDecays("h0", 1);
h = BosonDecays.at("h0_1");
h = BosonDecays.at(25, 0);
h = BosonDecays.at("h0", 1);
```

In this example, a `DecayTable` is created for holding the decays of SM bosons. The decays of the individual channels, each an instance of the `DecayTable::Entry` subclass, are then added to the table with the `()` method. A branching fraction for $W \to e \, \nu_e$ is then extracted directly from the `DecayTable`. The entire entry for Higgs decays is then extracted in six different ways, showing the standard bracket methods, the constant `at` access method (reminiscent of regular C++ STL container access), and the three different ways of referring to the particle whose decays are desired from the table.

The `DecayTable` machinery integrates seamlessly with SLHA and its generalisations, via `SLHAstruct` SLHAea objects. Individual `DecayTable::Entry` instances can be constructed directly from SLHA DECAY blocks provided in `SLHAea::Block` format, and made to emit them with the method `getSLHAea_block`. Likewise, a complete `DecayTable` can be constructed directly from an SLHA file

or an `SLHAstruct` object containing DECAY blocks, and can be output to either format with `writeSLHA` and `getSLHAea`.

Full documentation of the programming interface provided by the `DecayTable` and `DecayTable::Entry` classes can be found in

```
Elements/include/gambit/Elements/decaytable.
hpp.
```

3.3.2 Utilities

The module functions `check_first_sec_gen_mixing` and `get_mass_es_pseudonyms` (Table 7) rely on a series of helper functions that probe the sfermionic MSSM mixing matrices provided by **SpecBit**, in order to determine the gauge composition of a given mass or family-specific mass eigenstate, or the converse for gauge eigenstates. These functions are most relevant for **DecayBit** and backends such as **SUSY-HIT**, but are needed also by **ColliderBit** and **DarkBit**. They therefore ship as part of the main **GAMBIT** distribution, in the `Elements` directory. Functions can be found here for returning:

– the mass eigenstate with largest contribution from a given gauge eigenstate, plus full mixing and composition information if required (`slhahelp::mass_es_from_gauge_es()`)
– the gauge eigenstate with largest contribution from a given mass eigenstate, plus full mixing and composition information if required (`slhahelp::gague_es_from_mass_es()`)
– the (6-flavour) mass eigenstate that best matches a given family (2-flavour) mass state, plus full mixing and composition information if required (`slhahelp::mass_es_closest_to_family()`)
– the family (2-flavour) mass state that best matches a given (6-flavour) mass eigenstate, plus full mixing and composition information if required (`slhahelp::family_state_closest_to_mass_es()`)
– the two (6-flavour) mass eigenstates that best match a requested generation, as well as the implied (2-flavour) family mixing matrix between the family states (`slhahelp::family_state_mix_matrix()`).

Full documentation can be found in

```
Elements/include/gambit/Elements/mssm_
slhahelp.hpp.
```

The **DecayBit** module proper contains just two utilities of note that are not actual module functions (in the **GAMBIT** sense).

The first is the class `DecayBit::mass_es_pseudonyms`, which is used as the return type for `get_mass_es_pseudonyms` after obtaining the necessary content from the `slhahelp` functions. This is essentially just a container of named strings, one for each SLHA1 MSSM weak/family eigenstate sfermion.

The second is the function `DecayBit::CP_conjugate()`, which takes as its lone argument a `DecayTable::Entry` and returns a corresponding `DecayTable::Entry` for the CP conjugate of the original particle, assuming CP symmetry. The X = `bar` and Y = `minus` variants of the module functions in Tables 4 and 6 consist of nothing more than a call to this function, using their **dependencies** on the corresponding X = empty and Y = `plus` variants.

3.3.3 A worked example: SM-like Higgs decays

To illustrate some of the workings of **DecayBit** more concretely, here we go through the example of decays of the Higgs in detail.

The rollcall header `DecayBit_rollcall.hpp` declares a series of functions that are each capable of generating a `DecayTable::Entry` containing the width and branching fractions for Higgs decays:

```
#define CAPABILITY
   Reference_SM_Higgs_decay_rates
START_CAPABILITY

  #define FUNCTION Ref_SM_Higgs_decays_table
  START_FUNCTION(DecayTable::Entry)
  DEPENDENCY(mh, double)
  #undef FUNCTION

#undef CAPABILITY

#define CAPABILITY Higgs_decay_rates
START_CAPABILITY

  #define FUNCTION SM_Higgs_decays
  START_FUNCTION(DecayTable::Entry)
  DEPENDENCY(Reference_SM_Higgs_decay_rates,
   DecayTable::Entry)
  #undef FUNCTION

  #define FUNCTION SingletDM_Higgs_decays
  START_FUNCTION(DecayTable::Entry)
  DEPENDENCY(Reference_SM_Higgs_decay_rates,
   DecayTable::Entry)
  DEPENDENCY(SingletDM_spectrum, Spectrum)
  ALLOW_MODELS(SingletDM, SingletDMZ3)
  #undef FUNCTION

  #define FUNCTION MSSM_h0_1_decays
  START_FUNCTION(DecayTable::Entry)
  DEPENDENCY(SLHA_pseudonyms,
   DecayBit::mass_es_pseudonyms)
  BACKEND_REQ(cb_widthhl_hdec, (sh_reqd),
   widthhl_hdec_type)
  BACKEND_REQ(cb_wisusy_hdec, (sh_reqd),
```

```
   wisusy_hdec_type)
  BACKEND_REQ(cb_wisfer_hdec, (sh_reqd),
   wisfer_hdec_type)
  BACKEND_OPTION( (SUSY_HIT), (sh_reqd) )
  ALLOW_MODELS(MSSM63atQ, MSSM63atMGUT)
  #undef FUNCTION

#undef CAPABILITY
```

The first of these, `Ref_SM_Higgs_decays_table`, calculates the width and decay branching fractions of the pure SM Higgs, interpolating in the tables of Ref. [110] using the Higgs pole mass (provided by another module function as a **dependency**). This decay information is presented to the rest of the code with **capability** `Reference_SM_Higgs_decay_rates`, to allow other functions to declare that they depend on knowing the pure SM decays as reference quantities, even if the *actual* decays of the Higgs in the model being scanned turn out to be different from the SM case. The function `SM_Higgs_decays` has a **dependency** on this reference set of decay information, and simply presents it 'as is' to the rest of the code as the definitive decays of the Higgs, as indicated by the **capability** `Higgs_decay_rates`. The module function `SingletDM_Higgs_decays` gives the Higgs decays in the case of an SM-like Higgs that can also decay to two singlets. This function also piggybacks off the SM reference decay information, simply adding in the $h \rightarrow SS$ decay and rescaling the SM branching fractions. `MSSM_h0_1_decays` returns decays of the lightest MSSM Higgs.

The latter two functions have corresponding `ALLOW_MODELS` declarations, whereas `SM_Higgs_decays` is considered to be universally compatible with all models, and it will be used as a fallback in any case where a model-specific Higgs decay function does not exist, i.e. assuming that the Higgs is unaltered from the SM. The singlet variant also has a **dependency** on the `SingletDM_spectrum`, which it uses to extract the S mass and coupling in order to compute the invisible width.

The MSSM variant has no **dependency** on a `Spectrum` object, because it instead relies on `BACKEND_REQ`uirements, which it demands to be filled from its only declared `BACKEND_OPTION`, SUSY-HIT. A spectrum object is not needed because the backend initialisation function for SUSY-HIT has this **dependency** itself, as that is where the pole masses, couplings and mixings are actually fed into SUSY-HIT. The function `MSSM_h0_1_decays` does have a **dependency** on the `SLHA_pseudonyms` however, as these are needed to interpret the weak/family eigenstate decay widths returned by SUSY-HIT, and to place them into the resulting `DecayTable::Entry` in terms of mass eigenstates.

The macro calls illustrated in these declarations are described in more detail in the full GAMBIT paper [11].

The SM reference function computes the total width and branching fractions of the Higgs as follows:

```
// Reference SM Higgs decays from Ref.
   [110].
void Ref_SM_Higgs_decays_table(
     DecayTable::Entry& result)
{
  using namespace
    Pipes::Ref_SM_Higgs_decays_table;

  // Get the Higgs pole mass
  double mh = *Dep::mh;

  // Invalidate point if m_h is outside the
   most
  // reliable part of the LHCHiggsXSWG [110]
  // tables.
  double minmass =
   runOptions->getValueOrDef<double>(90.0,
   "higgs_minmass");
  double maxmass =
   runOptions->getValueOrDef<double>(160.0,
   "higgs_maxmass");
  if (mh < minmass or mh > maxmass)
{ std::stringstream msg; msg << "Requested
   Higgs virtuality is " << mh << ";
   allowed range is " << minmass << "--" <<
   maxmass << " GeV.";
   invalid_point().raise(msg.str()); }

  // Set the contents of the Entry
  result.calculator = "GAMBIT::DecayBit";
  result.calculator_version =
   gambit_version;
  result.width_in_GeV =
   virtual_SMHiggs_widths
     ("Gamma",mh);
  result.set_BF(virtual_SMHiggs_widths
               ("bb",mh),
               0.0, "b", "bbar");
...
}
```

First the Higgs mass is extracted from the `mh` **dependency**, and checked against the allowed minimum and maximum Higgs mass. These bounds are used to ensure that the SM-like Higgs is not outside the range over which the values covered by the tables of Ref. [110] are most reliable. The calculator and version are then set, and the total width is extracted from the tables of Ref. [110] with the function `virtual_SMHiggs_widths` (found in `Elements/src/virtual_higgs.cpp`). Finally, branching fractions to different final states are set by calls to `set_BF()`, in a similar fashion to the first example of Sect. 3.3.1.

The body of `SM_Higgs_decays` simply rebrands this information from `Reference_SM_Higgs_decay_rates` to `Higgs_decay_rates`, indicating to the rest of GAMBIT that it indeed represents the true Higgs decays in this case, and should be used as such in later calculations:

```
// SM decays: Higgs
void SM_Higgs_decays (DecayTable::Entry&
  result)
{
  namespace Pipes::SM_Higgs_decays;
  result =
   *Dep::Reference_SM_Higgs_decay_rates;
}
```

The decays of the Higgs are harvested and assembled with the decays of all other particles into a single `DecayTable`, by the function `all_decays`:

```
#define CAPABILITY decay_rates
START_CAPABILITY

  #define FUNCTION all_decays
  START_FUNCTION(DecayTable)
  DEPENDENCY(Higgs_decay_rates,
   DecayTable::Entry)
  ...
  MODEL_CONDITIONAL_DEPENDENCY(h0_2_decay_
    rates,DecayTable::Entry, MSSM63atQ,
    MSSM63atMGUT)
  ...
  #undef FUNCTION

#undef CAPABILITY
```

Here we see the generic **dependency** upon `Higgs_decay_rates`; regardless of the model being scanned, `all_decays` requires a `DecayTable::Entry` that describes the decays of the Higgs boson. Depending on which model is actually being scanned, this can end up being fulfilled by any one of the three functions with **capability** `Higgs_decay_rates` described above. The function `all_decays` also has a `MODEL_CONDITIONAL_DEPENDENCY` on the set of decays of each BSM particle, such as the second CP-even Higgs in the MSSM. The function body of `all_decays` simply takes all the entries that it depends on, and adds them to a single `DecayTable` object, along the lines of the second example in Sect. 3.3.1.

3.3.4 Adding support for new models and programs

Supporting a new model in DecayBit is a matter of:

1. writing additional module functions that can generate `DecayTable::Entry` objects for the novel field content,
2. adding new module functions that can compute new decays of existing particles in the new model, and
3. adding **dependencies** on these new functions to `all_decays`.

Adding a new backend capable of calculating decay rates and branching fractions is essentially the same as adding any new backend, and should follow the recipe presented in the main GAMBIT paper [11]. After this, using the new backend

from DecayBit requires writing new module decay functions designed to take advantage of it, or modifying existing ones to do so. In some rare cases, the backended functions might have the same signature as those of an existing backend, in which case the backend simply needs to be selected at the YAML file level, using the `Rules` section (see Sect. 6 of [11]).

4 PrecisionBit

The PrecisionBit module serves a number of related purposes. It provides so-called nuisance likelihoods, which describe the uncertainties on known quantities that are important inputs to other calculations, which one might like to vary within their experimentally allowed ranges in the course of a scan. This functionality is discussed in Sect. 4.1. Examples include the mass of the top quark and the strong coupling; similar quantities relevant only to dark matter observables are dealt with in DarkBit (e.g. the strange quark content of the proton and the local density of dark matter).

PrecisionBit is also responsible for calculating BSM corrections to precision SM observables, such as the mass of the W boson and the weak mixing angle θ_W, and providing correspondingly corrected pole masses and couplings to the rest of GAMBIT wherever appropriate. It also provides likelihood functions for quantifying the agreement of the predicted corrections to the precision observables with experimental data. We discuss PrecisionBit's ability to compute MSSM precision corrections and likelihoods in Sect. 4.2. Except for precision mass corrections, PrecisionBit does not (yet) support the calculation of precision corrections in non-supersymmetric models; these will be implemented as relevant when the corresponding models are implemented in GAMBIT.

Figure 9 shows how these different components of PrecisionBit fit together, and how the two 'halves' of the module are connected via SpecBit and the input SM parameters.

4.1 Standard model nuisances

PrecisionBit provides a number of SM nuisance likelihoods, related to SM couplings, quark masses and the masses of the Z and W bosons. These likelihoods are generally approximated by χ^2-like functions, comparing experimentally determined central values (and errors) with the SM input values provided by GAMBIT. Whenever they are available and relevant, we add theoretical errors to the experimental ones in quadrature (i.e. assuming that they are uncorrelated).[18]

---

[18] Some members of the LHC Higgs Cross-section Working Group advocate a linear combination (see Sect. 12.5 of [115]), on the basis of an assumed flat constraint on the theoretical systematic. However, no statistical justification is given for this, and as far as we can see it can only
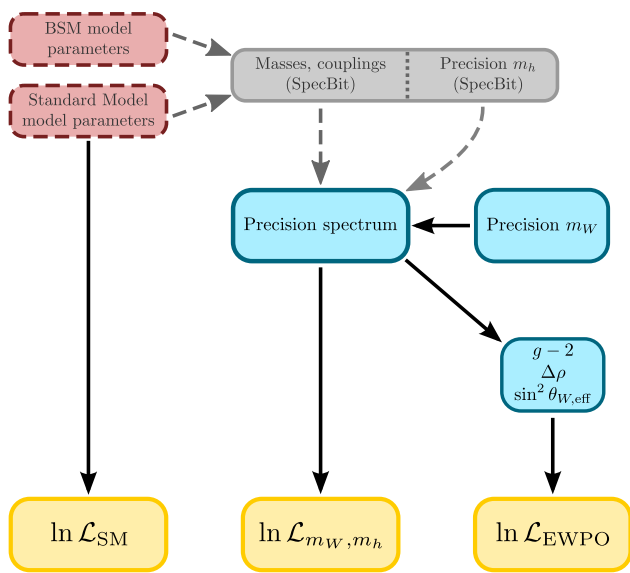
Fig. 9 Schematic representation of the structure of PrecisionBit. Model parameters (red boxes) are used by SpecBit to provide masses and couplings (grey box) to PrecisionBit, and they are also used by PrecisionBit directly to compute most SM nuisance likelihoods (left yellow box). PrecisionBit uses the precision Higgs masses from SpecBit (grey box right), along with its own precision calculation of the W mass (upper-right blue box), to convert the spectrum provided by SpecBit into a precision-updated version (upper-left blue box). It then uses the improved spectrum for calculating likelihoods associated with the Higgs and W masses (middle yellow box), as well as a suite of electroweak precision observables (EWPO) (lower blue box). It then feeds the EWPO predictions to its own experiment-based EWPO likelihoods (right yellow box)

4.1.1 Couplings

Likelihoods associated with the Fermi coupling constant (G_F) and fine-structure constant (α_{em}) are constructed by comparing model input values with those determined from electroweak global fits, with $G_F = (1.1663787 \pm 0.0000006) \times 10^{-5}$ GeV^{-2} and $\alpha_{em}(m_Z)^{-1} = 127.940 \pm 0.014$ ($\overline{MS}$ scheme) [61]. Similarly, the strong coupling (α_s) likelihood is calculated using a χ^2-averaged value of $\alpha_s(m_Z) = 0.1185 \pm 0.0005$ ($\overline{MS}$ scheme) taken from the sub-field of lattice determinations [61]. Uncertainties are taken to correspond to 1σ confidence intervals with no additional theoretical uncertainties. The PrecisionBit functions providing these likelihoods are summarised in Table 8.

4.1.2 Masses

Likelihoods associated with the $\overline{MS}$ light quark (u, d, s) masses at $\mu = 2$ GeV, $m_c(m_c)$, and $m_b(m_b)$, are calculated by PrecisionBit by comparing input values from GAMBIT with the corresponding PDG values and uncertainties [61], approximating the likelihoods as χ^2 functions. Similarly, the likelihood corresponding to the top quark mass is evaluated using combined Tevatron and LHC measurements [116]. For the light quark masses a single, joint likelihood is calculated with terms corresponding to the ratio of u and d masses, the ratio of the s mass to the sum of u and d masses, and m_s, respectively. Users can optionally provide their own values for these three quantities.

The W and Z boson mass likelihoods are based on mass measurements and corresponding uncertainties from the Tevatron and LEP experiments [61]. These strictly correspond to the mass parameter in a Breit–Wigner function rather than pole masses, but the difference is negligible for almost all purposes. PrecisionBit also provides a simple Higgs-mass likelihood, using the ATLAS and CMS combination [117] and is used directly as the PDG average [118]. The module functions providing the W, Z and h mass likelihoods are summarised in Table 8. Unlike the other SM particle masses, the model-dependent W and h mass **dependencies** are satisfied by functions specific to different model spectrum types, listed in Table 9. In this case, if theoretical uncertainties are available, we add them in quadrature to the experimental uncertainties when evaluating the χ^2 approximation to the mass likelihoods. By summing in quadrature, we assume that the experimental and theoretical errors are uncorrelated, which is reasonable given that they come from entirely different sources. The more detailed Higgs likelihoods of ColliderBit [14] will usually be more useful than the simple Higgs-mass likelihood provided in DecayBit. The latter can offer a lightweight alternative when varying the mass of the SM Higgs as a nuisance parameter in simple scans, as in e.g. Ref. [18].

4.2 MSSM precision observables

PrecisionBit allows for users to calculate a collection of precision observables using an array of external code packages. These observables can be accessed directly through module functions, passed as input to likelihood calculations, or used to update particle spectra for use throughout GAMBIT.

4.2.1 External code interfaces

Currently, PrecisionBit makes use of the external code packages FeynHiggs [22], SuperIso [33] and GM2Calc [35] for calculating precision observables in the MSSM. The GAMBIT interface to FeynHiggs was described already in

Table 8 PrecisionBit module functions providing log-likelihood calculations related to SM couplings and particle masses. All functions depend on `SMINPUTS` with the exception of the likelihoods of the W and h masses, which depend directly on the calculated W and h masses, respectively. These are available from both SM and BSM spectra, as summarised in Table 9. Except in very simple scenarios, the more detailed Higgs likelihoods of **ColliderBit** [14] are generally to be preferred to `lnL_h_mass_chi2`, as they simultaneously take account of correlated constraints on the mass and couplings

Capability	Function (return type): brief description	Dependencies	Options (type)
`lnL_alpha_em`	`lnL_alpha_em_chi2(double)`: Computes the log-likelihood of the fine-structure constant, $\alpha_{\rm em}(m_Z)$, in the $\overline{MS}$ scheme	`SMINPUTS`	
`lnL_alpha_s`	`lnL_alpha_s_chi2(double)`: Computes the log-likelihood of the strong coupling constant, $\alpha_s(m_Z)$, in the $\overline{MS}$ scheme	`SMINPUTS`	
`lnL_GF`	`lnL_GF_chi2(double)`: Computes the log-likelihood of the Fermi coupling constant, G_F	`SMINPUTS`	
`lnL_light_quark_masses`	`lnL_light_quark_masses_chi2(double)`: Computes the joint log-likelihood of the $\overline{MS}$ masses of the u, d and s quarks at $\mu = 2$ GeV	`SMINPUTS`	`mud_obs(double)` `mud_obserr(double)` `msud_obs(double)` `msud_obserr(double)` `ms_obs(double)` `ms_obserr(double)`
`lnL_mcmc`	`lnL_mcmc_chi2(double)`: Computes the log-likelihood of the $\overline{MS}$ mass $m_c(m_c)$	`SMINPUTS`	
`lnL_mbmb`	`lnL_mbmb_chi2(double)`: Computes the log-likelihood of the $\overline{MS}$ mass $m_b(m_b)$	`SMINPUTS`	
`lnL_t_mass`	`lnL_t_mass_chi2(double)`: Computes the log-likelihood of the top quark pole mass	`SMINPUTS`	
`lnL_Z_mass`	`lnL_Z_mass_chi2(double)`: Computes the log-likelihood of the Z boson pole mass	`SMINPUTS`	
`lnL_W_mass`	`lnL_W_mass_chi2(double)`: Computes the log-likelihood of the W boson pole mass	`mw`	
`lnL_h_mass`	`lnL_h_mass_chi2(double)`: Computes the log-likelihood of the h boson pole mass	`mh`	

Sect. 3.1.3, and the interface to **SuperIso** is covered in detail in the **FlavBit** paper [15].

The module function `GM2C_SUSY` calls library routines from **GM2Calc** to calculate the anomalous magnetic moment of the muon and translates the result and the errors from a_μ to $(g-2)_\mu$. **GM2Calc** can accept parameters and masses given in SLHA conventions. The module function exploits **GAMBIT** helper functions, discussed in Sect. 3.3.2, to convert between SLHA and SLHA2, which **GAMBIT** uses internally to store the spectrum. The inputs from the spectrum object, including the SMINPUTS parameters, are then passed on to the **GM2Calc** object by the module func-

tion `GM2C_SUSY`. One important subtlety is that **GM2Calc** also has two non-SLHA inputs, which provide input values for the fine-structure constant, α, in the Thompson limit and another α, which includes SM fermion contributions to the on-shell photon vacuum polarisation. These definitions differ from the $\overline{MS}$ $\alpha(m_Z)$, the inverse of which appears in entry 1 of the SLHA input block `SMINPUTS`. By default, **GAMBIT** uses the default values recommended by the **GM2Calc** authors. It is possible to overwrite these values using the **YAML** options `GM2Calc_extra_alpha_e_MZ` and `GM2Calc_extra_alpha_e_thompson_limit`.

Table 9 Summary of PrecisionBit module functions providing the W and h masses from different GAMBIT Spectrum inputs. These functions have no options nor backend requirements

Capability	Function (return type): brief description	Dependencies
mw	mw_from_SM_spectrum (triplet\<double>): Provides the W mass and its uncertainties from an SM spectrum	SM_spectrum
	mw_from_SS_spectrum (triplet\<double>): Provides the W mass and its uncertainties from a scalar-singlet model spectrum	SingletDM_spectrum
	mw_from_MSSM_spectrum (triplet\<double>): Provides the W mass and its uncertainties from an MSSM spectrum	MSSM_spectrum
mh	mh_from_SM_spectrum (triplet\<double>): Provides the h mass and its uncertainties from an SM spectrum	SM_spectrum
	mh_from_SS_spectrum (triplet\<double>): Provides the h mass and its uncertainties from a scalar-singlet model spectrum	SingletDM_spectrum
	mh_from_MSSM_spectrum (triplet\<double>): Provides the mass and associated uncertainties of the most SM-like Higgs boson in the MSSM	MSSM_spectrum SMlike_Higgs_PDG_code

PrecisionBit also contains an experimental interface to the proprietary code SUSY-POPE [119,120], but this is not recommended for general use. In addition, a new FeynHiggs routine FHEWPO, based on work used for Refs. [121,122], has very recently been released for beta testing in Feyn-Higgs 2.13.0-beta. We understand that this beta is now to be considered production-stable,[19] and so expect to support it in a future version of GAMBIT. Further calculations of the EWPO may in the future become available through the Flexi-bleTools collaboration behind FlexibleSUSY or through use of auto-generation software, which is now being developed with GAMBIT.

The various interface functions to different external code packages within PrecisionBit are summarised in Table 10, and include calculations of backend-specific data-types for input to other functions, accessor functions for specific observables, and functions for updating particle spectra to include precision corrections to the W and Higgs-sector masses.

4.2.2 Electroweak precision observable likelihoods

Electroweak precision observables are well measured observables from the electroweak sector of the SM. Due to the precise measurements they can provide constraints on BSM physics. For a pedagogical introduction see Ref. [123], where

Z-pole and non Z-pole sets of electroweak precision observables are listed in Tables 1 and 2, respectively. Precision-Bit includes a collection of functions for using the precision observables calculated by external code packages to calculate likelihoods. Currently, likelihood functions are provided for the effective leptonic weak mixing angle $\sin^2 \theta_{W,\text{eff}}$, the departure from 1 of the ratio of the Fermi constants implied by neutral and charged currents $\Delta\rho$, and the muon anomalous magnetic moment a_μ (also referred to as $g - 2 \equiv 2a_\mu$). For each of these observables, PrecisionBit calculates likelihoods by combining experimental and theoretical uncertainties in quadrature (i.e. assuming that theoretical and experimental errors are not correlated), and constructing likelihood functions based on the difference between the calculated and experimentally measured values. The module functions that provide these **capabilities** are listed in Table 11.

PrecisionBit also offers two module functions able to provide the predicted SM contribution to $g - 2$. These are listed along with the likelihood functions in Table 11. The first is calibrated to e^+e^- data, giving $a_{\mu,\text{SM}} = (11659180.2 \pm 4.9) \times 10^{-10}$ [124], whereas the second comes from $\tau^+\tau^-$ data, and corresponds to $a_{\mu,\text{SM}} = (11659189.4 \pm 5.4) \times 10^{-10}$ [124].

For the $g - 2$ likelihood, we combine errors from the predicted SM contribution in quadrature with the theoretical error arising from the BSM contribution, as these two theory errors are entirely independent of each other, and therefore uncorrelated. In the likelihood function, we compare

---

[19] T. Hahn, personal communication

Table 10 PrecisionBit module function interfaces to external code packages for calculation of precision observables. These functions include calculations of backend-specific data-types for input to other functions, retrieval functions for specific observables, and functions for updating particle spectra for use throughout GAMBIT

Capability	Function (return type): brief description	Dependencies	Backend requirements
FH_Precision	FH_PrecisionObs (fh_PrecisionObs): Calculate precision observables in the MSSM with FeynHiggs	FH_Couplings_output	FeynHiggs
deltarho	FH_precision_deltarho(double): Retrieve parameter $\Delta\rho$ from FeynHiggs precision calculations	FH_Precision	
prec_mw	FH_precision_mw(double): Retrieve W mass in the MSSM from FeynHiggs precision calculations	FH_Precision	
prec_sinW2_eff	FH_precision_sinW2(double): Retrieve effective leptonic weak mixing angle $\sin^2\theta_{W,\mathrm{eff}}$ from FeynHiggs precision calculations	FH_Precision	
edm_e	FH_precision_edm_e(double): Retrieve electron electric dipole moment from FeynHiggs precision calculations	FH_Precision	
edm_n	FH_precision_edm_n(double): Retrieve neutron electric dipole moment from FeynHiggs precision calculations	FH_Precision	
edm_hg	FH_precision_edm_hg(double): Retrieve mercury electric dipole moment from FeynHiggs precision calculations	FH_Precision	
MSSM_spectrum	make_MSSM_precision_ spectrum_H_W(Spectrum): Function to provide an updated MSSM spectrum with precision W and Higgs masses	unimproved_MSSM_spectrum prec_mw prec_HiggsMasses	
	make_MSSM_precision_ spectrum_W(Spectrum): Function to provide an updated MSSM spectrum with precision W mass	unimproved_MSSM_spectrum prec_mw	
	make_MSSM_precision_spectrum _none(Spectrum): Function to present an unimproved MSSM spectrum as if it is a precision spectrum	unimproved_MSSM_spectrum	
muon_gm2	FH_precision_gm2(double): Retrieve MSSM contribution to $g-2$ from FeynHiggs precision calculations	FH_Precision	
	GM2C_SUSY(double): Calculate MSSM contribution to $g-2$ using GM2Calc	MSSM_spectrum	GM2Calc
	SI_muon_gm2(double): Calculate MSSM contribution to $g-2$ using SuperIso	SuperIso_modelinfo	SuperIso
SP_PrecisionObs	SP_PrecisionObs(double): Calculate precision observables in the MSSM with SUSY-POPE (experimental)		SUSY-POPE

Table 11 PrecisionBit module functions providing log-likelihood calculations for electroweak precision observables

Capability	Function (return type): brief description	Dependencies
`lnL_sinW2_eff`	`lnL_sinW2_eff_chi2(double)`: Computes the log-likelihood associated with $\sin^2\theta_{W,\mathrm{eff}}$	`prec_sinW2_eff`
`lnL_gm2`	`lnL_gm2_chi2(double)`: Computes the log-likelihood associated with muon $g-2$	`muon_gm2` `muon_gm2_SM`
`lnL_deltarho`	`lnL_deltarho_chi2(double)`: Computes the log-likelihood associated with $\Delta\rho$	`deltarho`
`muon_gm2_SM`	`gm2_SM_ee(triplet<double>)`: Computes the SM contribution to $g-2$, based on e^+e^- data	
	`gm2_SM_tautau(triplet<double>)`: Computes the SM contribution to $g-2$, based on $\tau^+\tau^-$ data	

the sum of the SM and BSM contributions to the experimental measurement $a_\mu = (11659208.9 \pm 6.3) \times 10^{-10}$ [125,126], where the error is the sum in quadrature of statistical (5.4×10^{-10}) and systematic (3.3×10^{-10}) uncertainties.

Theoretical calculations of the MSSM contribution to a_μ can be taken from either GM2Calc, SuperIso or Feyn-Higgs. While SuperIso and FeynHiggs include essentially the same one and two-loop corrections, and give more or less consistent results, GM2Calc includes additional two-loop contributions and an improved on-shell calculation of the one-loop effects, leading to significant improvements in precision for some parameter combinations. GM2Calc calculates an uncertainty on its prediction, using the magnitudes of the two-loop Barr–Zee corrections [127] to estimate the magnitude of neglected higher-order contributions [35]; PrecisionBit adopts this estimate directly when employing GM2Calc. For FeynHiggs and SuperIso, no such estimate is available. Following the discussion in Ref. [128], we assign a theoretical uncertainty of either 30% or 6×10^{-10} (whichever is greater) to the values of $g-2$ that PrecisionBit obtains from SuperIso and FeynHiggs.

PrecisionBit's theoretical predictions of $\sin^2\theta_{W,\mathrm{eff}}$ and $\Delta\rho$ in the MSSM are currently provided exclusively by Feyn-Higgs. We assume a theoretical error, $\sigma^{\mathrm{theo}}_{\sin^2\theta_{W,\mathrm{eff}}}$, of 12×10^{-5} on $\sin^2\theta_{W,\mathrm{eff}}$ [129]. The fractional MSSM corrections to both $\sin^2\theta_{W,\mathrm{eff}}$ and m_W are approximately proportional to $\Delta\rho$ at tree level, with a constant of proportionality quite close to one in magnitude (within a few tens of percent [129]). Combined with the fact that $\Delta\rho \ll 1$ in general, to a good approximation the theoretical error on its value can be estimated from the fractional uncertainty on $\sin^2\theta_{W,\mathrm{eff}}$ or m_W; we therefore adopt a theory error on $\Delta\rho$ of

$$\sigma^{\mathrm{theo}}_{\Delta\rho} = \max\left(\sigma^{\mathrm{theo}}_{m_W}/m_W, \, \sigma^{\mathrm{theo}}_{\sin^2\theta_{W,\mathrm{eff}}}/\sin^2\theta_{W,\mathrm{eff}}\right), \qquad (14)$$

where $\sigma^{\mathrm{theo}}_{m_W} = 10\,\mathrm{MeV}$ [129].

4.2.3 Precision-updated MSSM spectrum

As described in Table 10, PrecisionBit includes a number of module function for updating existing MSSM spectra with precision calculations of particle masses:

```
make_MSSM_precision_spectrum_H_W(),
make_MSSM_precision_spectrum_W(), and
make_MSSM_precision_spectrum_none().
```

These functions take an `unimproved_MSSM_spectrum` as a dependency, update its mass spectrum with precision values, and return the improved version as an `MSSM_spectrum`. The `H_W` version updates both the Higgs-sector and the W masses, the `W` version does only the W mass, and `make_MSSM_precision_spectrum_none` simply transmutes an `unimproved_MSSM_spectrum` into an `MSSM_spectrum` without further modification. The precision W and Higgs-sector masses are typically provided via **dependencies** fulfilled by functions that call FeynHiggs (though they can of course in principle come from anywhere).

In addition to updating the masses, the `make_MSSM_precision_spectrum_H_W()` and `make_MSSM_precision_spectrum_W()` functions also update the associated theoretical uncertainties provided by the original spectrum object. The uncertainty on the pole mass of the W is adopted from its **dependency** on the precision value of m_W. When calculating this with FeynHiggs, we assume an uncertainty

of 10 MeV [129] (as discussed in the context of the theory error on $\Delta\rho$ in Sect. 4.2.2).[20]

For the MSSM Higgs bosons (A^0, h^0, H^0 and $H^\pm$), the methods by which PrecisionBit arrives at the central value and uncertainties on the mass are configurable by the user, using the options `Higgs_predictions_source` and `Higgs_predictions_error_method`, respectively. For the `Higgs_predictions_source`, there are three possible values:

1: Update each of the Higgs masses to the value derived from the precision calculator (canonically FeynHiggs). This is the default.
2: Keep the masses unchanged from the original input spectrum.
3: Assign the mean of the masses found in the original spectrum and the ones provided by the precision calculator.

Five options are available for assigning theoretical uncertainties to the MSSM Higgs masses. These are based on different combinations of three quantities with unique values for each Higgs state: the error associated with the mass from the original input spectrum (Δ_s), the error determined by the precision calculator (Δ_p), and the difference of central values from the two calculations (Δ_g). All but one of the options for using these quantities assigns them as additional errors beyond a so-called 'range around chosen central' (RACC) calculation. The RACC is defined such that the upper error σ_+ is the distance from the central value chosen with `Higgs_predictions_error_method` to the greater of the values predicted by the original spectrum and the precision calculator. The lower error σ_- is defined analogously as the distance from the central value chosen to the lesser of the two predictions. With these definitions, the available options for the `Higgs_predictions_error_method` are:

1: Upper and lower uncertainties both set equal to the sum in quadrature of all three uncertainties: $\sigma_+^2 = \sigma_-^2 = \Delta_s^2 + \Delta_p^2 + \Delta_g^2$.
2: RACC, with Δ_s added to the error associated with the distance to the spectrum generator value (the spectrum-generator 'edge'), and Δ_p added to the precision-calculator edge. This is the default.
3: RACC, with $\Delta_g/2$ added to both σ_+ and σ_-.
4: RACC, with $\Delta_g/2$ added at the spectrum-generator edge, and Δ_p added at the precision-calculator edge.
5: RACC, with $\Delta_g/2$ added at the precision-calculator edge, and Δ_s added at the spectrum-generator edge.

---

[20] It is, however, worth noting that this is in fact no smaller than the value that we assign to the uncertainty on the *W* mass calculation performed by FlexibleSUSY, so the replacement has no effect if FlexibleSUSY has been used to generate the original `unimproved_MSSM_spectrum`.

The resulting precision-improved spectrum is provided to the rest of GAMBIT for physics calculations, and in the case of the Higgs bosons, computation of Higgs-mass likelihoods by ColliderBit.

5 Examples

5.1 Example YAML files

In `yaml_files`, we give some example input YAML files that run various functions from SpecBit, DecayBit and PrecisionBit within GAMBIT. The file `SpecBit_MSSM.yaml` runs a number of tests on the various MSSM spectra, using either FlexibleSUSY or SPheno from within SpecBit. The file `SpecBit_vacuum_stability.yaml` does a short (10 minutes runtime on a single core) scan of the Higgs and top mass, in order to map out the regions of SM vacuum (meta-)stability and perturbativity. The DecayBit example `DecayBit_MSSM20.yaml` computes decay widths and branching fractions for a single example point in the MSSM20atQ model [11]. `DecayBit_SingletDM` does the same for ten randomly selected parameter combinations in the scalar-singlet dark matter model. `PrecisionBit_MSSM 20.yaml` computes all precision observables and likelihoods contained in PrecisionBit, for the same example MSSM20atQ point as `DecayBit_MSSM20.yaml`.

5.2 3-BIT-HIT

As a convenience for users simply wishing to use some basic functionality of SpecBit, DecayBit and PrecisionBit from the command line, we also provide an additional driver program 3-BIT-HIT. This program doubles as an example of how the three modules can be used in standalone mode, i.e. without the GAMBIT Core or the other modules.

3-BIT-HIT takes a single MSSM model as input via a minimal YAML file `3bithit.in`, in either the weak-scale MSSM20atQ or GUT-scale NUHM2 parametrisation. The user can also specify SM parameters, following SLHA2 conventions. The program then solves the RGEs and evaluates pole masses, calculates decay widths and branching fractions of all SM particles and their superpartners, computes electroweak precision observables, and finally outputs the results as a single SLHA file `3bithit.out.slha`.

The basic spectrum generation in 3-BIT-HIT is done using FlexibleSUSY via SpecBit, with the Higgs and *W* masses replaced by precision calculations performed with Feyn-Higgs by PrecisionBit. The resulting SM-like Higgs mass and uncertainties emitted in the final SLHA file are based on the default settings in `PrecisionBit::make_MSSM_precision_spectrum_H_W()`:

```
Higgs_predictions_source = 1
Higgs_predictions_error_method = 2
```

The Higgs-sector decays are computed by DecayBit using FeynHiggs, SM decays are based on native DecayBit functions, and sparticle decays come from SDECAY, via SUSY-HIT and DecayBit. 3-BIT-HIT computes the SUSY contribution to the anomalous magnetic moment of the muon, along with its uncertainty, using calls to GM2Calc from Precision-Bit. All other electroweak precision observables (EWPO) come from FeynHiggs via PrecisionBit, where errors are also assigned to them.

3-BIT-HIT performs a similar function to SUSY-HIT, but computes EWPO in addition to spectra and decays. It also employs FlexibleSUSY instead of SuSpect for spectrum generation, FeynHiggs for Higgs decays rather than HDE-CAY, and our patched version of SDECAY (see Sect. 3.1.3 for details) for sparticle decays. Its source can be found in `DecayBit/examples/3bithit.cpp`, and it can be built with

```
make 3bithit
```

6 Summary

In this paper we have introduced the GAMBIT modules SpecBit, DecayBit and PrecisionBit. These are designed to flexibly integrate publicly available programs for spectrum generation, the calculation of decay widths and branching ratios, and additional precision calculations, like the anomalous magnetic moment of the muon. Together, these modules provide a powerful way to synergise spectrum generators, decay codes and additional precision calculations, allowing users to extract the information produced by them in a common format. We have illustrated this use with the example standalone program 3-BIT-HIT (Sect. 5), which provides a single interface to FlexibleSUSY, FeynHiggs, SDECAY and GM2Calc. At the same time, these modules play a crucial role within the full framework of GAMBIT, where they provide essential information to other packages and important components of the log-likelihood functions used to drive global fits.

Appendices

A Physics models

Here we give a compact physical definition of each of the models currently supported by SpecBit, DecayBit and PrecisionBit, complementing the slightly more code-focussed description in the main GAMBIT paper [11]. GAMBIT is primarily designed for global fits to extensions of the Standard Model (SM) of particle physics. Many calculations in these extensions require SM quantities, and many such extensions make minimal modifications to the SM; examples are the scalar-singlet dark matter model (Appendix A.3) and effective weakly interacting massive particle (WIMP) dark matter (see Ref. [13] for an example in GAMBIT). We therefore begin with the SM itself.

A.1 Standard model

The SM describes the interactions of all observed fundamental particles. These interactions are invariant under the symmetries of the gauge group,

$$G_{SM} = SU(3)_C \times SU(2)_L \times U(1)_Y. \tag{A.1}$$

The Lagrangian density is given by

$$
\begin{aligned}
\mathcal{L}_{SM} = &\sum_{i=1,3} i(\overline{Q}_i \not{D} Q_i + \overline{L}_i \not{D} L_i + \overline{u}_{Ri} \not{D} u_{Ri} \\
&+ \overline{d}_{Ri} \not{D} d_{Ri} + \overline{e}_{Ri} \not{D} e_{Ri}) \\
&+ \sum_{i,j=1,3} \left[\sum_{a,b=1,2} (\mathbf{Y}_u)_{ij} \epsilon_{ab} Q_i^a (\phi^\dagger)^b u_{Rj} \right. \\
&\left. + (\mathbf{Y}_d)_{ij} \overline{Q}_i \phi d_{Rj} + (\mathbf{Y}_e)_{ij} \overline{L}_i \phi e_{Rj} + h.c. \right]
\end{aligned}
$$

$$+ |D_\mu \phi|^2 - \mu^2 |\phi|^2 - \lambda |\phi|^4$$
$$- \frac{1}{4} B_{\mu\nu} B^{\mu\nu} - \frac{1}{4} W^A_{\mu\nu} W^{\mu\nu A} - \frac{1}{4} G^B_{\mu\nu} G^{\mu\nu B} \quad \text{(A.2)}$$

where u_{Ri}, d_{Ri} and e_{Ri} represent the right-handed up-type quark, down-type quark and lepton. Similarly,

$$\phi = \begin{pmatrix} \phi^+ \\ \phi^0 \end{pmatrix}, \quad Q_i = \begin{pmatrix} u_{Li} \\ d_{Li} \end{pmatrix}, \quad L_i = \begin{pmatrix} \nu_{Li} \\ e_{Li} \end{pmatrix}, \quad \text{(A.3)}$$

represent the Higgs, left-handed quark and left-handed lepton doublets, respectively. Here i and j specify the generation of the fields, a and b are $SU(2)_L$ indices, and ϵ_{ab} is the antisymmetric tensor.

The $B_{\mu\nu}$, $W^A_{\mu\nu}$ and $G^B_{\mu\nu}$ are field strength tensors for the B_μ, W^A_μ and G^B_μ fields respectively, which are, respectively, associated with the $U(1)_Y$, $SU(2)_L$ and $SU(3)_C$ gauge groups. The kinetic terms for the matter fields include $\slashed{D} = D^\mu \gamma_\mu$, where D_μ is the usual covariant derivative, which, for example, takes the following form for all fields transforming non-trivially under both $SU(2)_L$ and $SU(3)_C$:

$$D_\mu = \partial_\mu - i g_1 Q_Y B_\mu - i g_2 W^A_\mu T^A - i g_3 \lambda^B G^B_\mu. \quad \text{(A.4)}$$

Here g_1 and Q_Y are the GUT-normalised gauge coupling and GUT-normalised charges[21] of $U(1)_Y$. Similarly g_2 and T^A are the $SU(2)_L$ gauge couplings and generators, respectively, and g_3 and λ^B are the $SU(3)_C$ gauge couplings and generators, respectively.

Electroweak symmetry is broken when the Higgs field develops a VEV, v,

$$\langle \phi \rangle = \frac{1}{\sqrt{2}} \begin{pmatrix} 0 \\ v \end{pmatrix}, \quad \text{(A.5)}$$

so that field may be rewritten as

$$\phi^0 = \frac{1}{\sqrt{2}} (v + h^0 + i G^0). \quad \text{(A.6)}$$

where h^0 is the physical Higgs boson and G^0 is the unphysical Goldstone boson that corresponds to the longitudinal mode for the massive Z^0 boson.

The SM VEV gives masses to the SM particles. The Z and W gauge bosons obtain tree-level masses

$$m_Z = \frac{1}{2} \sqrt{(g'^2 + g_2^2)}\, v \quad M_W = \frac{1}{2} g_2 v. \quad \text{(A.7)}$$

---

[21] Hypercharge is often written without GUT normalisation, such that the charge for a field ϕ is $Y^\phi = \sqrt{5/3} Q^\phi_Y$ and will be written in the covariant derivative with the gauge coupling $g' = \sqrt{3/5} g_1$.

The weak mixing angle θ_W is given by

$$\sin^2 \theta_W \equiv \frac{g'^2}{g'^2 + g_2^2} = 1 - \frac{M_W^2}{m_Z^2}, \quad \text{(A.8)}$$

though it is important to note that the last equality depends on the tree-level relations given in Eq. (A.7), so this definition holds only within the renormalisation scheme in which the gauge couplings are defined. An alternative definition can be made in terms of pole masses,

$$\sin^2 \theta_W^{\text{on-shell}} \equiv 1 - \frac{M_W^{(\text{pole})\,2}}{m_Z^{(\text{pole})\,2}}. \quad \text{(A.9)}$$

Finally, one should note that neither of these correspond exactly to the leptonic effective weak mixing angle, which is an important precision observable. This quantity is relevant to PrecisionBit and is discussed in Sect. 4.2.2.

A.2 Minimal supersymmetric standard model

The MSSM superpotential is given by

$$\hat{W} = \epsilon_{ab} \left\{ \sum_{i,j=1,3} \left[(\mathbf{Y}_u)_{ij} \hat{Q}^a_i \hat{H}^b_u \hat{U}^c_j - (\mathbf{Y}_d)_{ij} \hat{Q}^a_i \hat{H}^b_d \hat{D}^c_j \right. \right.$$
$$\left. \left. - (\mathbf{Y}_e)_{ij} \hat{L}^a_i \hat{H}^b_d \hat{E}^c_j \right] - \mu \hat{H}^a_u \hat{H}^b_d \right\}, \quad \text{(A.10)}$$

where a hat indicates a superfield and i, j specify the generation, while a, b are $SU(2)_L$ indices. This includes the Higgsino mixing parameter μ, which appears as a dimension one coefficient of the up- and down-type Higgs superfields,

$$\hat{H}_u = \begin{pmatrix} \hat{H}^+_u \\ \hat{H}^0_u \end{pmatrix} \quad \hat{H}_d = \begin{pmatrix} \hat{H}^0_d \\ \hat{H}^-_d \end{pmatrix}. \quad \text{(A.11)}$$

The chiral superfields have dimensionless up- and down-type Yukawa couplings, $\mathbf{Y}_u$ and $\mathbf{Y}_d$ with the Higgs superfields and the chiral superfields for the right-handed quarks. Specifically left-handed quarks, $\hat{Q}$, interact with both the chiral superfields for the right-handed up quarks, $\hat{U}^c$, and the up-type Higgs superfield through $\mathbf{Y}_u$ and with down quarks, $\hat{D}^c$, and the down-type Higgs superfields through $\mathbf{Y}_d$. Similarly the chiral superfield for left-handed leptons $\hat{L}$ has Yukawa interactions, $\mathbf{Y}_e$ with the chiral superfields for the right-handed leptons, $\hat{E}^c$ and $\hat{H}_d$.

Terms which break supersymmetry without introducing quadratic divergences are given in the soft SUSY-breaking Lagrangian,

$$\mathcal{L}_{\text{soft}} = -\frac{1}{2}\left[M_1\bar{\tilde{B}}^0\tilde{B}^0 + M_2\bar{\tilde{W}}_A\tilde{W}_A + M_3\bar{\tilde{g}}_b\tilde{g}_b\right] \quad \text{(A.12a)}$$

$$-\frac{i}{2}\left[M_1'\bar{\tilde{B}}^0\gamma_5\tilde{B}^0 + M_2'\bar{\tilde{W}}_A\gamma_5\tilde{W}_A + M_3'\bar{\tilde{g}}_B\gamma_5\tilde{g}_B\right]$$
$$\text{(A.12b)}$$

$$-\left[bH_u^a H_{da} + \text{h.c.}\right] - m_{H_u}^2|H_u|^2 - m_{H_d}^2|H_d|^2$$
$$\text{(A.12c)}$$

$$+\sum_{i,j=1,3}\left\{-\left[\tilde{Q}_i^\dagger(\mathbf{m}_Q^2)_{ij}\tilde{Q}_j + \tilde{d}_{Ri}^\dagger(\mathbf{m}_d^2)_{ij}\tilde{d}_{Rj}\right.\right.$$
$$\left.+\tilde{u}_{Ri}^\dagger(\mathbf{m}_u^2)_{ij}\tilde{u}_{Rj} + \tilde{L}_i^\dagger(\mathbf{m}_L^2)_{ij}\tilde{L}_j + \tilde{e}_{Ri}^\dagger(\mathbf{m}_e^2)_{ij}\tilde{e}_{Rj}\right]$$
$$\text{(A.12d)}$$

$$+-\epsilon_{ab}\left[(\mathbf{T}_u)_{ij}\tilde{Q}_i^a H_u^b \tilde{u}_{Rj}^\dagger - (\mathbf{T}_d)_{ij}\tilde{Q}_i^a H_d^b \tilde{d}_{Rj}^\dagger\right.$$
$$\left.-(\mathbf{T}_e)_{ij}\tilde{L}_i^a H_d^b \tilde{e}_{Rj}^\dagger + \text{h.c.}\right]$$
$$\text{(A.12e)}$$

$$-\epsilon_{ab}\left[(\mathbf{C}_u)_{ij}\tilde{Q}_i^a H_d^{*b}\tilde{u}_{Rj}^\dagger - (\mathbf{C}_d)_{ij}\tilde{Q}_i^a H_u^{*b}\tilde{d}_{Rj}^\dagger\right.$$
$$\left.\left.-(\mathbf{C}_e)_{ij}\tilde{L}_i^a H_u^{*b}\tilde{e}_{Rj}^\dagger + \text{h.c.}\right]\right\}.$$
$$\text{(A.12f)}$$

As in the previous equation we have indices i, j for generation and a, b for $SU(2)_L$ and in addition $A \in \{1, 2, 3\}$ is the index for $SU(2)_L$ gauge generators and $B \in \{1, 2, 3 \ldots 8\}$ is the index for the $SU(3)_C$. The superpartners of the $U(1)_Y$ gauge field, $\tilde{B}^0$, the $SU(2)_L$ gauge fields, $\tilde{W}_A$, and the $SU(3)_C$ gauge fields, $\tilde{g}_B$ get soft masses, M_1, M_2 and M_3, respectively, as well as potential contributions from M_1', M_2' and M_3'. The Higgs scalar fields, H_u and H_d receive soft scalar mass contributions, $m_{H_u}^2$, $m_{H_d}^2$, respectively, and they are mixed by the soft bilinear mass squared, b. The left-handed squarks, $\tilde{Q}_i$ and sleptons, $\tilde{L}_i$ and the right-handed up-type squarks, $\tilde{u}_{Ri}$, down-type squarks, $\tilde{d}_{Ri}$, and charged sleptons $\tilde{e}_{Ri}$ all have soft scalar masses, $(\mathbf{m}_Q^2)_{ij}$, $(\mathbf{m}_L^2)_{ij}$, $(\mathbf{m}_u^2)_{ij}$, $(\mathbf{m}_d^2)_{ij}$ and $(\mathbf{m}_e^2)_{ij}$, respectively. Finally we have soft trilinears for the up-type squarks and Higgs, $(\mathbf{T}_u)_{ij}$, down-type squarks and Higgs, $(\mathbf{T}_d)_{ij}$, and the sleptons and down-type Higgs, $(\mathbf{T}_e)_{ij}$ and also the terms that are soft when there is no singlet in the model, $(\mathbf{C}_u)_{ij}$, $(\mathbf{C}_d)_{ij}$ and $(\mathbf{C}_e)_{ij}$. The latter are normally assumed to be zero, even though the MSSM has no scalar singlets as it is difficult to generate these. The soft trilinears may also be rewritten in terms of the following:

$$(\mathbf{A}_f)_{ij} = \frac{(\mathbf{T}_f)_{ij}}{(\mathbf{Y}_f)_{ij}} \quad f \in \{u, d, e\}. \quad \text{(A.13)}$$

In the MSSM the neutral components of the up- and down-type scalar Higgs fields, H_u^0 and H_d^0, respectively, develop VEVs,

$$\langle H_d \rangle = \frac{1}{\sqrt{2}}\begin{pmatrix} v_d \\ 0 \end{pmatrix}, \quad \langle H_u \rangle = \frac{1}{\sqrt{2}}\begin{pmatrix} 0 \\ v_u \end{pmatrix}, \quad \text{(A.14)}$$

and the bilinear soft term, b, mixes them, so that they must be rotated to obtain the physical mass eigenstates, h_i

$$\begin{pmatrix} h_1 \\ h_2 \end{pmatrix} = \sqrt{2}\,Z_H\begin{pmatrix} \mathcal{R}e(H_d^0) - \frac{v_d}{\sqrt{2}} \\ \mathcal{R}e(H_u^0) - \frac{v_u}{\sqrt{2}} \end{pmatrix}, \quad \text{(A.15)}$$

where the mixing matrix is commonly expressed in terms of a mixing angle α,

$$Z_H = \begin{pmatrix} -\sin\alpha & \cos\alpha \\ \cos\alpha & \sin\alpha \end{pmatrix}. \quad \text{(A.16)}$$

Similarly the pseudoscalar Higgs state, A, is formed from the imaginary parts of the Higgs scalar fields

$$\begin{pmatrix} G^0 \\ A^0 \end{pmatrix} = \sqrt{2}\,Z_A\begin{pmatrix} \mathcal{I}m(H_d^0) \\ \mathcal{I}m(H_u^0) \end{pmatrix}. \quad \text{(A.17)}$$

where G^0 is the neutral Goldstone boson. At tree level it is straightforward to show that the mass eigenstates can be obtained with a rotation,

$$Z_A = \begin{pmatrix} -\sin\beta & \cos\beta \\ \cos\beta & \sin\beta \end{pmatrix} \quad \text{(A.18)}$$

giving a massless Goldstone boson and an expression for the pseudoscalar mass,

$$m_A^2 = \frac{b}{\sin\beta\cos\beta}. \quad \text{(A.19)}$$

The charged Higgs-mass eigenstate, H^+ and charged Goldstone boson, G^+ are related to the charged scalar Higgs fields through

$$\begin{pmatrix} G^+ \\ H^+ \end{pmatrix} = Z_{H^+}\begin{pmatrix} H_u^+ \\ (H_d^-)^\dagger \end{pmatrix}. \quad \text{(A.20)}$$

The squarks are rotated into the mass eigenstates as,

$$\begin{pmatrix} \tilde{u}_1 \\ \tilde{u}_2 \\ \tilde{u}_3 \\ \tilde{u}_4 \\ \tilde{u}_5 \\ \tilde{u}_6 \end{pmatrix} = Z_{\tilde{u}}\begin{pmatrix} \tilde{Q}_1^1 \\ \tilde{Q}_2^1 \\ \tilde{Q}_3^1 \\ \tilde{u}_{R1}^\dagger \\ \tilde{u}_{R2}^\dagger \\ \tilde{u}_{R3}^\dagger \end{pmatrix}, \quad \begin{pmatrix} \tilde{d}_1 \\ \tilde{d}_2 \\ \tilde{d}_3 \\ \tilde{d}_4 \\ \tilde{d}_5 \\ \tilde{d}_6 \end{pmatrix} = Z_{\tilde{d}}\begin{pmatrix} \tilde{Q}_1^2 \\ \tilde{Q}_2^2 \\ \tilde{Q}_3^2 \\ \tilde{d}_{R1}^\dagger \\ \tilde{d}_{R2}^\dagger \\ \tilde{d}_{R3}^\dagger \end{pmatrix} \quad \text{(A.21)}$$

and the sleptons and sneutrinos are rotated as,

$$
\begin{pmatrix} \tilde{e}_1 \\ \tilde{e}_2 \\ \tilde{e}_3 \\ \tilde{e}_4 \\ \tilde{e}_5 \\ \tilde{e}_6 \end{pmatrix} = Z_{\tilde{e}} \begin{pmatrix} \tilde{L}_1^2 \\ \tilde{L}_2^2 \\ \tilde{L}_3^2 \\ \tilde{e}_{R1}^\dagger \\ \tilde{e}_{R2}^\dagger \\ \tilde{e}_{R3}^\dagger \end{pmatrix}, \quad \begin{pmatrix} \tilde{\nu}_1 \\ \tilde{\nu}_2 \\ \tilde{\nu}_3 \end{pmatrix} = Z_{\tilde{\nu}} \begin{pmatrix} \tilde{L}_1^1 \\ \tilde{L}_2^1 \\ \tilde{L}_3^1 \end{pmatrix}. \tag{A.22}
$$

The neutralinos are mass eigenstates from combinations of the Bino $\tilde{B}^0$, neutral Wino $\tilde{W}_3$ and the up- and down-type Higgsinos, $\tilde{H}_u^0$ and $\tilde{H}_d^0$,

$$
\begin{pmatrix} \tilde{\chi}_1^0 \\ \tilde{\chi}_2^0 \\ \tilde{\chi}_3^0 \\ \tilde{\chi}_4^0 \end{pmatrix} = Z_N \begin{pmatrix} \tilde{B}^0 \\ \tilde{W}_3 \\ \tilde{H}_d^0 \\ \tilde{H}_u^0 \end{pmatrix}, \tag{A.23}
$$

where we follow the SLHA [1] convention and require the mixing matrix, Z_N, to be real, which means the masses of the neutralinos may be negative. Note that this is the form of the masses and mixings that can be accessed from the spectrum object in GAMBIT and used for passing into interfaces to various backend codes that need the spectrum as an input. We follow the SLHA convention of negative masses because this ensures the maximum number of codes can be supported without conversion during interfacing. For any codes or internal calculations which expect neutralino masses and mixings in another convention, this must be converted in the interface.

The mass matrix of the charginos is diagonalised through a biunitary transformation, such that the two component chargino mass eigenstates are given by

$$
\begin{pmatrix} \chi_1^+ \\ \chi_2^+ \end{pmatrix} = U_+ \begin{pmatrix} \tilde{W}_1 - i\tilde{W}_2 \\ \tilde{H}_u^+ \end{pmatrix}, \tag{A.24}
$$

$$
\begin{pmatrix} \chi_1^- \\ \chi_2^- \end{pmatrix} = U_- \begin{pmatrix} \tilde{W}_1 + i\tilde{W}_2 \\ \tilde{H}_u^- \end{pmatrix} \tag{A.25}
$$

A.3 Scalar-singlet dark matter model

The scalar-singlet model is a minimalistic extension of the SM, with one new scalar charged only under a $\mathbb{Z}_2$ symmetry. This symmetry guarantees that the scalar is a stable dark matter candidate, and restricts the most general permitted renormalisable Lagrangian density on symmetry arguments to be

$$
\mathcal{L}_{SS} = \frac{1}{2}\mu_S^2 S^2 + \frac{1}{2}\lambda_{hS} S^2 |H|^2 + \frac{1}{4}\lambda_S S^4 + \frac{1}{2}\partial_\mu S \partial^\mu S, \tag{A.26}
$$

where μ_S^2 is a mass squared term for the singlet, λ_{hS} the Higgs-portal coupling and λ_S the quartic self coupling.

The singlet mass has an additional contribution from EWSB through the Higgs-portal coupling, giving a mass,

$$
m_S = \sqrt{\mu_S^2 + \frac{1}{2}\lambda_{hS} v^2}, \tag{A.27}
$$

where v is the SM Higgs VEV in Eq. (A.5). As the $\mathbb{Z}_2$ symmetry must remain unbroken for stable dark matter we demand that the scalar field, S, does not obtain a VEV and thus no mixing occurs between the Higgs and the new scalar field. This requirement is satisfied for $\lambda_S > 0$.

B Spectrum generators as backends: SPheno

One of the key features of GAMBIT is its ability to interface with different external codes to perform relevant calculations for a specific scan. These codes, or backends, are linked with GAMBIT in a non-trivial way in order to access their functions directly via memory, rather than through a shell. Details and descriptions of the backend system and how to create a new backend can be found in the main GAMBIT paper [11].

Consequently, as with other calculations, the generation of a spectrum can be delegated to a backend library which provides the required capability. At the time of the first release, GAMBIT comes along with FlexibleSUSY [19] internally implemented as the de facto spectrum generator for MSSM models, but also allows the use of the backended spectrum generator SPheno [20,21]. In this section we describe how SPheno is added as a backend to GAMBIT and how it provides the spectrum capability `unimproved_MSSM_spectrum`, to serve as a guide for the addition of other spectrum generators in the future, e.g. SOFTSUSY [36–39] or SuSpect [40]. It is worth noting that SPheno is written in Fortran, so some of the details of the backending process will differ significantly from software written in other languages; SOFTSUSY, for example, is written in C++, so it would need to draw on the classloading tool **BOSS** (Backend-On-a-Stick Script, described in [11]).

B.1 Installation and import of variables/functions from the backend

The general guideline to create a GAMBIT backend from an external tool, as described in the main GAMBIT paper [11], is to compile it into a shared library in order to provide access to the internal variables and functions at runtime. This thus requires patching the makefile provided by SPheno to create this shared library. Effectively one needs to add the flag `-shared` to the rule `lib/libspheno.so`, as well as the flag `-fPIC` to the compilation rule for every source file.

Once compiled, this makes the variables and functions inside SPheno available to use in GAMBIT. However,

the main Fortran program `SPheno` does not get a shared library symbol, and thus all the functions and subroutines inside, such as the one that calculates the spectrum `CalculateSpectrum`, are not made available to GAMBIT. This is easily fixed by a further patch into the main file `SPheno3.f90`, to change the `Program` directive for a `Module` directive, and commenting all the executed lines in the program, but leaving the subroutines unscathed.

Therefore, all the variables and functions can be accessed by the backend system. These are imported by using the macros `BE_VARIABLE` and `BE_FUNCTION`, described in the Backends section of the GAMBIT main paper [11], with the library symbols `__module_MOD_var` for a variable `var` in a module `module`. Note that Fortran is case insensitive, but all library symbols are written in small caps (except the MOD tag). The files `Backends/include/gambit/Backends/frontends/SPheno.hpp` and `SpecBit/src/frontends/SPheno.cpp` are then created to import these variables and functions, and to give any specifications that are necessary.

Along with the imported variables and functions, a set of convenience functions is created, by the macro `BE_CONV_FUNCTION`, see Ref. [11], specified in the frontend file `SPheno.cpp`. These are `run_SPheno`, `ReadingData`, `InitializeStandardModel`, `Spectrum_Out` and `ErrorHandling`. In particular the convenience function `run_SPheno` will effectively be the main function of the backend and will cover the executed lines commented out by the patch, avoiding those that perform input and output operations, which will be described in the section below.

Lastly, every backend gets a backend initialisation function, where any overall initialisations for the backend are performed, inside the namespace `BE_INI_FUNCTION` in the frontend source file. The only initialisation required by SPheno at this stage is the information about the model which is done by the function `ModelInUse`.

B.2 Input and output, warnings and errors

SPheno follows the SLHA conventions [1,2] for the format of the spectra it generates, and as such all its input and (successful) output happens in the form of files using the SLHA format. Reading and writing from files is inefficient and hence, for the most part, GAMBIT avoids performing any of these tasks at the point level, for it would slow down the scans significantly. Therefore, all the input required and output provided by SPheno is stored and obtained directly to and from the backend variables, as described above, via two convenience functions `ReadingData` and `Spectrum_Out`.

The convenience function `ReadingData` initialises all required SPheno variables for the relevant model and scan. Firstly all the SM parameters are initialised in the convenience function `InitializeStandardModel` using the structure `SMInputs`, followed by some internal initialisation

of variables via the backended functions `InitializeLoopFunctions` and `Set_All_Parameters_0`. In addition all the run options, as described in detail in Sect. 2.2.6, are parsed at this step.

Secondly, all the information pertaining to the SLHA blocks MINPAR, EXTPAR, MSL2, MSE2, MSQ2, MSU2, MSD2, TUIN, TDIN and TEIN is pulled from the model parameters and stored in the corresponding SPheno variables. As expected, not every variable is required to be filled for every model, e.g. for the CMSSM, only the variables relating to the MINPAR block are filled, which are obtained from the model parameters `M0`, `M12`, `TanBeta`, `SignMu` and `A0`. Variables relating to other SLHA input blocks are either filled somewhere else, SMINPUTS in `InitializeStandardModel`, and MODSEL in the backend initialisation function, as described above; or are ignored for they have no internal use as an input, such as SPINFO.

The output convenience function `Spectrum_Out` deals with the creation of a `Spectrum` object from the spectrum generated by SPheno. As was mentioned in Sect. 2.2.4, the backended SPheno version does not provide post-generation RGE running, like FlexibleSUSY does. Rather, the spectrum is stored in a `SLHAstruct` variable and then transformed into a static `Spectrum` object via the function `spectrum_from_SLHAea`. By default the scale dependent output (SLHA blocks MSOFT, GAUGE, etc.) is given at the SUSY scale.

Warnings and error messages in SPheno are also written into a file and, likewise, GAMBIT diverts that output to avoid slowing down the scans. Internally this is done by initializing the variable `ErrCan` to 0, hereby forcing SPheno to write any output into an unused buffer and not to a file. Error tracking is hence done by use of the integer variable `kont`, whose value depends on the specific error that occurred. For any non-zero value of `kont`, the convenience function `ErrorHandling` appends the corresponding message to the logger and raises an `invalid_point` exception. Specific details on which values of `kont` correspond to which error messages can be found in Appendix C of the SPheno manual [21].

B.3 Calculation of the spectrum

The spectrum is calculated in SPheno by using the back-ended function `CalculateSpectrum`, provided all required variables are initialised previously. After this function is run, all the output variables are filled (including the error tracking variable `kont`), and they are written into a `Spectrum` object, as described above.

Once the spectrum is calculated, it is returned to the function that had this backend requirement, by providing the capability `SPheno_MSSMspectrum`. The built-in function with this backend requirement is `get_MSSM_spectrum_SPheno`. This function provide the capability `unimproved_MSSM_`

Table 12 List of available `SubSpectrumContents` definitions

Name	Plain English description	Tables
SM	SM pole masses and QED×QCD running parameters	16
SM_slha	SM pole masses and SLHA SMINPUTS MSbar masses (light quarks)	16
SMHiggs	SM Higgs sector; pole mass and VEV	16
MSSM	63 parameter general MSSM in SLHA2 conventions	17
SingletDM	Scalar-singlet dark matter plus SM Higgs sector	18

spectrum, effectively passing along the created `Spectrum` object to the next level in the dependency tree. The definition of the function `get_MSSM_spectrum_SPheno` is

```
 #define FUNCTION get_MSSM_spectrum_SPheno
START_FUNCTION(Spectrum)
ALLOW_MODELS(CMSSM, MSSM63atMGUT,
  MSSM63atQ)
DEPENDENCY(SMINPUTS, SMInputs)
BACKEND_REQ(SPheno_MSSMspectrum,
  (libSPheno), int, (Spectrum&, const
  Finputs&) )
BACKEND_OPTION((SPheno, 3.3.8),
  (libSPheno))
 #undef FUNCTION
```

where `Finputs` is a data structure containing the input parameters and the run options.

All of the additional calculations available in the out-of-the-box version of SPheno, such as low-energy observables, branching ratios and cross sections, are disabled in this backended version, as their functionality is already covered by other GAMBIT modules: FlavBit [15], DecayBit and ColliderBit [14].

C List of SpecBit `SubSpectrumContents` definitions

Spectrum information that has been stored in SpecBit may be accessed via the `SubSpectrum` interface class. The stored information is given in Table 16 (for low-energy SM information), Table 17 (for the MSSM) and Table 18 (for the scalar-singlet model). To extract this information SpecBit has accessors that have one of the following forms:

```
subspectrum.get(tag, label)
subspectrum.get(tag, label, index 1)
subspectrum.get(tag, label, index 1, index 2)
```

The arguments *label* and *tag* are given in Tables 16, 17 and 18 in columns 3 and 4, respectively. The fifth column in these tables specifies the number of indices that are required as additional arguments, which will be either 0, 1 or 2. When necessary the last two columns in these tables provide the range each index runs over. These indices specify the element in a vector or matrix of parameters that are structured this way.

If the `using namespace Par` directive is not in use in the scope, then *tag* must be prefixed with the namespace qualifier `Par`, e.g. `Par::`*tag*.

The `SubSpectrum` classes provide interfaces to specific backend spectrum generators, while `SubSpectrumContents` classes define what parameters a `SubSpectrum` wrapper must provide. In Table 12 we list the `SubSpectrumContents` definitions which ship with the SpecBit module in GAMBIT 1.0.0, and describe each of them in detail in the table referenced beside each list item.

D List of SpecBit `SubSpectrum` wrappers

Here we provide a list of all the `SubSpectrum` wrapper classes which ship with the GAMBIT 1.0.0 version of SpecBit. Wrappers based on simple parameter containers with no RGE facilities are listed in Table 13, while wrappers that connect to full spectrum calculators are listed in Table 14. The interface to the wrapped content of these classes is defined by their associated `SubSpectrumContents` classes, which are given in the "Contents" column of each table. The `SubSpectrumContents` classes are themselves described in Appendix C.

E `SubSpectrum` wrapper `GetterMaps` and `SetterMaps` function signatures

In this appendix we give details of the allowed function pointer signatures for the `GetterMaps` and `SetterMaps` objects which constitute the link between `SubSpectrum` interface functions, and backend spectrum data (see Appendix F.2). In principle the system can be extended to allow any function signature. However, it cannot be done automatically, and thus at present only a limited set of options is available. If the object to which you want to interface does not conform to one of these signatures, you will need to write 'helper' functions in your wrapper which alter the function signature. Some of the function signatures listed are designed to support such helper functions.

The list is given in Table 15. So, for example, to store a function pointer with the signature `double f(Model&)` in a

Table 13 List of available "simple" `SubSpectrum` wrappers, which wrap only parameter data and provide no RGE facilities, and they are therefore not connected to any backend spectrum generators. Spectrum data is either entered directly as input parameters or retrieved from an external source, such as an SLHA file. The final column 'Contents' gives the `SubSpectrumContents` definition (see Appendix C) to which the wrapper conforms

Name	Plain English description	Contents
`MSSMSimpleSpec`	General container for MSSM spectrum. Can be constructed from an SLHAea object	`MSSM`
`SLHASimpleSpec`	Base class for wrappers based on SLHAea	n/a
`SMHiggs SimpleSpec`	Very simple wrapper for SM Higgs parameters	`SMHiggs`
`SMSimpleSpec`	General container for SM spectrum data. Can be constructed from an SLHAea object	`SM_slha`
`ScalarSingletDM SimpleSpec`	Simple wrapper for scalar-singlet DM parameters, plus SM gauge and Yukawa couplings	`SingletDM`

Table 14 List of available "full" `SubSpectrum` wrappers, which connect directly to backend spectrum generator code and provide RGE-running facilities. The final column 'Contents' gives the `SubSpectrumContents` definition (see Appendix C) to which the wrapper conforms

Name	Plain English description	Contents
`MSSMSpec`	General templated wrapper for all FlexibleSUSY MSSM models	`MSSM`
`QedQcdWrapper`	Wraps QED×QCD calculator from SOFTSUSY	`SM`
`SingletDMSpec`	Wraps FlexibleSUSY spectrum for scalar-singlet model	`SingletDM`

Table 15 List of data members of the `GetterMaps` and `SetterMaps` classes (which are `std::map`s), and the corresponding signatures of function pointers which can be stored in those maps. Note, however, that functions requiring index input cannot be stored directly in the map; they must first be associated with sets of allowed values for those indices using the listed "Helper class"; see text. In the table, the name `Self` refers to the `SubSpectrum` wrapper class itself (that is, these maps accept function pointers to members functions of the wrapper class)

Data member name	Type	Helper class	"Getter" function signature	"Setter" function signature
`map0`	`fmap0`	none	`double Model::f()`	`void Model::f(double)`
`map1`	`fmap1`	`FInfo1`	`double Model::f(int)`	`void Model::f(int,double)`
`map2`	`fmap2`	`FInfo2`	`double Model::f(int,int)`	`void Model::f(int,int,double)`
`map0W`	`fmap0W`	none	`double Self::f()`	`void Self::f(double)`
`map1W`	`fmap1W`	`FInfo1W`	`double Self::f(int)`	`void Self::f(double,int)`
`map2W`	`fmap2W`	`FInfo2W`	`double Self::f(int,int)`	`void Self::f(double,int,int)`
`map0_extraM`	`fmap0_extraM`	none	`double f(Model&)`	`void f(Model&,double)`
`map1_extraM`	`fmap1_extraM`	`FInfo1M`	`double f(Model&,int)`	`void f(Model&,double,int)`
`map2_extraM`	`fmap2_extraM`	`FInfo2M`	`double f(Model&,int,int)`	`void f(Model&,double,int,int)`
`map0_extraI`	`fmap0_extraI`	none	`double f(Input&)`	`void f(Input&,double)`
`map1_extraI`	`fmap1_extraI`	`FInfo1I`	`double f(Input&,int)`	`void f(Input&,double,int)`
`map2_extraI`	`fmap2_extraI`	`FInfo2I`	`double f(Input&,int,int)`	`void f(Input&,double,int,int)`

`GetterMaps` collection, one needs to put it into the `map0` map, as shown in the example in Appendix F.3. The `map0W` (see Appendix F.2), `map0_extraM` and `map0_extraI` maps work similarly, with function signatures as shown in the table.

Functions associated with indices require an extra step; they must be associated with lists of the allowed indices. This is made easy via the `FInfo` helper classes, which are listed in Table 15. For example, say we wish to attached two-index

functions with the signature `double Model::f(int,int)` to the `get` interface. The `fill_getter_maps` function for a wrapper *Wrap* would then be:

```
static Wrap::GetterMaps
  Wrap::fill_getter_maps()
{
  static const std::set<int> i012 =
    initSet(0,1,2);
```

Table 16 The spectrum contents available via the get function for the wrappers conforming to the SM and SM_slha (Standard Model) contents definitions (Appendix F.1), except the Higgs pole mass and VEV, which are available from wrappers conforming to the SMHiggs definition and which contain *only* those two parameters. We found this split to be convenient since BSM models commonly alter the Higgs sector. The above quantities may all be obtained via *subspectrum*.get(Par::*tag,label,index1,index2*) where *label* and *tag* are given in columns 3 and 4, respectively, and *subspectrum* is defined as in Sect. 2.3.3. The final two arguments are optional and are filled when necessary by the indices from the allowed range given in columns 6 and 7. *These indices only apply to SM contents; for SM_slha wrappers only u_1, d_1, and d_2 are available. The scale at which all $\overline{MS}$ parameters are defined is returned by *subspectrum*.GetScale() (see Sect. 2.3.3 and Fig. 6), unless the wrapper object does not support RGE running, in which case the scales should be in accordance with SLHA2 SMINPUTS definitions (this is the convention followed in the SMSimpleSpec wrapper class, for example. See Appendix D for full lists of the currently available wrapper classes). Note that if definitions in accordance with SLHA2 SMINPUTS are desired, one can always access the parameters in the SMInputs structure of the parent Spectrum object rather than performing the required RGE running; for example *spectrum*.get_SMInputs().mS retrieves the strange quark $\overline{MS}$ mass at 2 GeV. See Fig. 7 for the definition of the SMInputs class, and see Figs. 3 and 4 for the structure of Spectrum objects. Note that the weak mixing angle, provided in the form $\sin^2\theta_W^{\rm pole}$, corresponds to the pole definition from Eq. (A.9).

Quantity	Plain English description	String	Tag	Number of indices	Index 1 range	Index 2 range
$m_\gamma^{\rm pole}$	Photon pole mass	"gamma"	Pole_Mass	0	n/a	n/a
$m_g^{\rm pole}$	Gluon pole mass	"g"	Pole_Mass	0	n/a	n/a
$m_Z^{\rm pole}$	Z pole mass	"Z0"	Pole_Mass	0	n/a	n/a
$m_W^{\rm pole}$	W pole mass	"W+"/"W-"	Pole_Mass	0	n/a	n/a
$m_t^{\rm pole}$	Top quark pole mass	"u_3"/"ubar_3"/"t"/"tbar"	Pole_Mass	0	n/a	n/a
$m_b^{\rm pole}$	Bottom quark pole mass	"d_3"/"dbar_3"/"b"/"bbar"	Pole_Mass	0	n/a	n/a
$m_e^{\rm pole}$	Electron pole mass	"e-"/"e+"	Pole_Mass	0	n/a	n/a
$m_\mu^{\rm pole}$	Muon pole mass	"mu-"/"mu+"	Pole_Mass	0	n/a	n/a
$m_\tau^{\rm pole}$	Tau pole mass	"tau-"/"tau+"	Pole_Mass	0	n/a	n/a
$m_{l_i}^{\rm pole}$	Charged lepton pole masses	"e-"/"e+"	Pole_Mass	1	1,2,3	n/a
$m_{\nu_i}^{\rm pole}$	Neutrino pole masses	"nu"/"nubar"	Pole_Mass	1	1,2,3	n/a
$\sin^2\theta_W^{\rm pole}$ in (A.9)	Weak mixing angle	"sinW2"	Pole_Mixing	0	n/a	n/a
α_e	Fine-structure constant	"alpha"	dimensionless	0	n/a	n/a
α_s	Strong coupling constant	"alphaS"	dimensionless	0	n/a	n/a
m_γ	Photon $\overline{MS}$ mass	"gamma"	mass1	0	n/a	n/a
m_g	Gluon $\overline{MS}$ mass	"g"	mass1	0	n/a	n/a
m_u	Up-type quark $\overline{MS}$ masses	"u"/"ubar"	mass1	1	1,2,3*	n/a
m_d	Down-type quark $\overline{MS}$ masses	"d"/"dbar"	mass1	1	1,2,3*	n/a
m_l	Charged lepton $\overline{MS}$ masses	"e-"/"e+"	mass1	1	1,2,3*	n/a
m_{h^0}	Higgs boson pole mass	"h0"	Pole_Mass	0	n/a	n/a
v in (A.5)	Higgs VEV	"vev"	mass1	0	n/a	n/a

Table 17 The spectrum contents available via the `get` function for wrappers conforming to the `MSSM` spectrum contents definition (Appendix F.1). These quantities may be obtained via *subspectrum*.get(`Par::`*tag*,*label*,*index 1*,*index 2*) where *label* and *tag* are given in columns 3 and 4, respectively, and *subspectrum* is defined as in Sect. 2.3.3. The final two arguments are optional and are filled when necessary by the indices from the allowed range given in the last two columns. Note that weak mixing angle here is defined in the $\overline{DR}$ scheme and is simply reconstructed from the gauge couplings using the expression given in Eq. (A.8)

Quantity	Plain English description	String	Tag	Index 1 range	Index 2 range
$m_{\tilde{g}}^{\text{pole}}$	Gluino pole masses	`"~g"`	`Pole_Mass`	n/a	n/a
$m_{W^\pm}^{\text{pole}}$	W boson pole mass	`"W+"`	`Pole_Mass`	n/a	n/a
$m_{A^0}^{\text{pole}}$	Pseudoscalar Higgs pole mass	`"A0"`	`Pole_Mass`	n/a	n/a
$m_{H^+}^{\text{pole}}$	Charged Higgs pole masses	`"H+"`	`Pole_Mass`	n/a	n/a
$m_{h^0}^{\text{pole}}$	CP-even Higgs pole masses	`"h0"`	`Pole_Mass`	{1,2}	n/a
$m_{\tilde{u}}^{\text{pole}}$	up-type squark pole masses	`"~u"`	`Pole_Mass`	{1,2,3,4,5,6}	n/a
$m_{\tilde{d}}^{\text{pole}}$	down-type squark pole masses	`"~d"`	`Pole_Mass`	{1,2,3,4,5,6}	n/a
$m_{\tilde{e}}^{\text{pole}}$	slepton pole masses	`"~e"`	`Pole_Mass`	{1,2,3,4,5,6}	n/a
$m_{\tilde{\nu}}^{\text{pole}}$	sneutrino pole mass	`"~nu"`	`Pole_Mass`	{1,2,3}	n/a
$m_{\chi^\pm}^{\text{pole}}$	chargino pole masses	`"~chi+"`	`Pole_Mass`	{1,2}	n/a
$m_{\chi^0}^{\text{pole}}$	neutralino pole masses	`"~chi0"`	`Pole_Mass`	{1,2,3,4}	n/a
$Z_{\tilde{u}}$ in (A.21)	up-type squark mixing	`"~u"`	`Pole_Mixing`	{1,2,3,4,5,6}	{1,2,3,4,5,6}
$Z_{\tilde{d}}$ in (A.21)	down-type squark mixing	`"~d"`	`Pole_Mixing`	{1,2,3,4,5,6}	{1,2,3,4,5,6}
$Z_{\tilde{e}}$ in (A.22)	slepton mixing	`"~e"`	`Pole_Mixing`	{1,2,3,4,5,6}	{1,2,3,4,5,6}
$Z_{\tilde{\nu}}$ in (A.22)	sneutrino mixing	`"~nu"`	`Pole_Mixing`	{1,2,3}	{1,2,3}
Z_N in (A.23)	neutralino mixing	`"~chi0"`	`Pole_Mixing`	{1,2,3,4}	{1,2,3,4}
U_+ in (A.25)	chargino mixing	`"~chi+"`	`Pole_Mixing`	{1,2}	{1,2}
U_- in (A.25)	chargino mixing	`"~chi-"`	`Pole_Mixing`	{1,2}	{1,2}
Z_H in (A.15)	CP even Higgs mixing	`"h0"`	`Pole_Mixing`	{1,2}	{1,2}
Z_A in A.17	CP odd Higgs mixing	`"A0"`	`Pole_Mixing`	{1,2}	{1,2}
$B\mu$ in (A.12c)	soft Higgs bilinear	`"BMu"`	`mass2`	n/a	n/a
$m_{H_u}^2$ in (A.12c)	soft Higgs-mass squared	`"mHu2"`	`mass2`	n/a	n/a
$m_{H_d}^2$ in (A.12c)	soft Higgs-mass squared	`"mHd2"`	`mass2`	n/a	n/a
m_A^2 in (A.19)	$\overline{DR}$ pseudoscalar mass squared	`"mA2"`	`mass2`	n/a	n/a
$\mathbf{m}_Q^2$ in (A.12d)	soft squark mass squared	`"mq2"`	`mass2`	{1,2,3}	{1,2,3}
$\mathbf{m}_u^2$ in (A.12d)	soft up-squark mass squared	`"mu2"`	`mass2`	{1,2,3}	{1,2,3}
$\mathbf{m}_d^2$ in (A.12d)	soft down-squark mass squared	`"md2"`	`mass2`	{1,2,3}	{1,2,3}
$\mathbf{m}_L^2$ in (A.12d)	soft slepton mass squared	`"ml2"`	`mass2`	{1,2,3}	{1,2,3}
$\mathbf{m}_e^2$ in (A.12d)	soft slepton mass squared	`"me2"`	`mass2`	{1,2,3}	{1,2,3}
M_1 in (A.12a)	Bino soft mass	`"M1"`	`mass1`	n/a	n/a
M_2 in (A.12a)	Wino soft mass	`"M2"`	`mass1`	n/a	n/a
M_3 in (A.12a)	Gluino soft mass	`"M3"`	`mass1`	n/a	n/a
Mu in (A.10)	Superpotential Higgs bilinear	`"Mu"`	`mass1`	n/a	n/a
vu in (A.14)	Up-type Higgs VEV	`"vu"`	`mass1`	n/a	n/a
vd in (A.14)	Down-type Higgs VEV	`"vd"`	`mass1`	n/a	n/a
$\mathbf{T}_u$ in (A.12e)	Trilinear	`"TYu"`	`mass1`	{1,2,3}	{1,2,3}
$\mathbf{T}_d$ in (A.12e)	Trilinear	`"TYd"`	`mass1`	{1,2,3}	{1,2,3}
$\mathbf{T}_e$ in (A.12e)	Trilinear	`"TYe"`	`mass1`	{1,2,3}	{1,2,3}
g_1 in (A.4)	$U(1)_Y$ gauge coupling[24]	`"g1"`	`dimensionless`	n/a	n/a

Table 17 continued

Quantity	Plain English description	String	Tag	Index 1 range	Index 2 range
g_2 in (A.4)	$SU(2)_L$ gauge coupling	`"g2"`	dimensionless	n/a	n/a
g_3 in (A.4)	$SU(3)_C$ gauge coupling	`"g3"`	dimensionless	n/a	n/a
$\tan\beta$	ratio of VEVs v_u/v_d	`"tanbeta"`	dimensionless	n/a	n/a
$\sin^2\theta_W$ in (A.8)	weak mixing angle	`"sinW2"`	dimensionless	n/a	n/a
$\mathbf{Y}_u$ in (A.10)	Up-type quark Yukawa	`"Yu"`	dimensionless	{1,2,3}	{1,2,3}
$\mathbf{Y}_d$ in (A.10)	Down-type quark Yukawa	`"Yd"`	dimensionless	{1,2,3}	{1,2,3}
$\mathbf{Y}_e$ in (A.10)	lepton Yukawa	`"Ye"`	dimensionless	{1,2,3}	{1,2,3}

```
GetterMaps map_collection;

map_collection[Par::tag].map2[label]
   = MTget::FInfo2(&Model::f,i012,i012)

return map_collection;
}
```

where `std::set<int> i012` is initialised with a helper function `initSet` (from the header `gambit/Utils/util_functions.hpp`). This set is then used to tell the `GetterMaps` that each index of the function can accept the values 0, 1 or 2. At present, it is not possible to restrict the allowed index values pair-wise (that is, GAMBIT cannot automatically provide safeguards against users asking for specific invalid combinations of indices).

For more examples of how to add functions to `GetterMaps` and `SetterMaps` please see the source code for the various `SubSpectrum` wrappers that ship with SpecBit. These are listed in Appendix D.

F Adding support for new spectrum calculators

While the GAMBIT modules described in this paper are initially distributed with the MSSM and scalar-singlet models already implemented, the great advantage of this framework is the manner in which new models can be systematically added. In this section we describe how to do this.

The most simple way to extend SpecBit is to just add a new `SubSpectrumContents` definition, which is described in Appendix F.1. This creates a new standardised set of string label/tag/index sets to which new `SubSpectrum` wrappers may choose to conform. This is the first step that is required for adding a wrapper for an entirely new model, but it does not actually provide any real **capabilities**, it only standardises the interface for wrappers which provide access to the same basic physics model.

The real work of interfacing to new spectrum information is done by writing a `SubSpectrum` wrapper, which is the standardised GAMBIT interface to any container of spectrum information. This information may be supplied using a

new spectrum generator created with FlexibleSUSY, a new model in SARAH / SPheno, or any available public or private spectrum generator for that model (which is written in C/C++ or Fortran; codes written in other languages can at present only be launched via calls to the operating system).

The new spectrum generator then needs to be interfaced with SpecBit. If the spectrum generator is written in C++ and has getters and setters to access the spectrum information (and a run method) then this task can be greatly simplified. It is also possible to use the spectrum objects as a simple container for parameters, with no RGE-running facilities (i.e. no actual spectrum generator "hooked up"), which is the simplest use case.

There are therefore a number of different ways that the Spectrum interface class can be connected to parameter data. We will tackle these various cases in order of increasing complexity throughout the following sections.

As well as designing the wrapper class, in order to actually use the new spectrum object one must write GAMBIT module functions that construct the objects and return them as GAMBIT **capabilities** (running the backend spectrum generator if required). This task will be discussed in Appendix F.7.

As a quick-reference for writers of new wrapper classes, in Sect. 2.4 we provide a checklist of tasks to be completed when writing a wrapper, with references to the subsection of this guide in which the relevant detailed instructions can be found.

F.1 SubSpectrumContents definitions

In this section we describe the mechanism which enforces the consistency of `SubSpectrum` objects wrapping different spectrum calculators. Recall that `SubSpectrum` objects are simply a virtual interface class; the real work of interfacing to external spectrum calculators is performed by wrapper classes which derive from `SubSpectrum`. In the process of defining this wrapper class, one must associate the wrapper with a `SubSpectrumContents` class. The `SubSpectrumContents` classes are simple `structs` that define which parameters are supposed to be contained by

Table 18 The spectrum contents available via the `get` function for wrappers conforming to the `SingletDM` (scalar-singlet) spectrum contents definition. These quantities may be obtained via `subspectrum.get(Par::tag,label,index 1,index 2)` where *label* and *tag* are given in columns 3 and 4, respectively, and *subspectrum* is defined as in Sect. 2.3.3. The final two arguments are optional and are filled when necessary by the indices from the allowed range given in columns 6 and 7

Quantity	Plain English description	String	Tag	Number of indices	Index 1 range	Index 2 range
m_S^{pole}	Scalar pole mass	`"s0"`	`Pole_Mass`	0	n/a	n/a
$m_{h^0}^{\text{pole}}$	Higgs pole mass	`"h0"`	`Pole_Mass`	0	n/a	n/a
v in (A.5)	Higgs VEV	`"vev"`	`mass1`	0	n/a	n/a
λ in (A.2)	Higgs quartic coupling	`"lambda_h"`	`dimensionless`	0	n/a	n/a
λ_s in (A.26)	Scalar quartic coupling	`"lambda_S"`	`dimensionless`	0	n/a	n/a
λ_{hs} in (A.26)	Higgs quartic coupling	`"lambda_hS"`	`dimensionless`	0	n/a	n/a
g_1 in (A.4)	$U(1)_Y$ gauge coupling	`"g1"`	`dimensionless`	0	n/a	n/a
g_2 in (A.4)	$SU(2)_W$ gauge coupling	`"g2"`	`dimensionless`	0	n/a	n/a
g_3 in (A.4)	$SU(3)_C$ gauge coupling	`"g3"`	`dimensionless`	0	n/a	n/a
$\sin^2\theta_W$ in Eq. (A.8)	weak mixing angle	`"sinW2"`	`dimensionless`	0	n/a	n/a
$\mathbf{Y}_u$ in (A.2)	Up-type quark Yukawa	`"Yu"`	`dimensionless`	2	{1, 2, 3}	{1, 2, 3}
$\mathbf{Y}_d$ in (A.2)	Down-type quark Yukawa	`"Yd"`	`dimensionless`	2	{1, 2, 3}	{1, 2, 3}
$\mathbf{Y}_e$ in (A.2)	lepton Yukawa	`"Ye"`	`dimensionless`	2	{1, 2, 3}	{1, 2, 3}

wrappers that interface to a calculator for a specific physics model. So, for example, two wrapper classes which wrap two different external spectrum calculators, but which calculate the spectrum for the same physics model, should both be associated with the same `SubSpectrumContents` class. The shared `SubSpectrumContents` class then acts as a promise that identical 'get' calls made to each of the two wrappers should retrieve equivalent information. Furthermore, if a wrapper fails to make certain parameters accessible that are defined as part of the `SpectrumContents`, then a runtime error will occur when the spectrum is constructed. The error will explain that the wrapper does not conform to the declared contents and so cannot be used. This message should serve only to help wrapper writers not to forget any contents; it will not occur when using completed wrappers.

The first release of **SpecBit** defines the set of `SubSpectrumContents` classes listed below:

```
namespace SpectrumContents {
 struct SM                 { SM(); };
 struct SM_slha            { SM_slha(); };
 struct SMHiggs            { SMHiggs(); };
 struct MSSM               { MSSM(); };
 struct SingletDM          { SingletDM(); };
}
```

and descriptions of each of them can be found in Appendix C. The above code shows the entire declaration for these classes, as can be found in `Models/include/gambit/Models/SpectrumContents/Regi- steredSpectra.hpp`, with the exception that each class is derived from a base class `SubSpectrumContents` (omitted here for brevity). The only definition required for these classes is their constructor, which for the above classes are defined in source files with matching names in the `Models/src/SpectrumContents/` directory. Defining the constructor, which defines the required parameters for that spectrum type, is very simple. It is easiest to explain with an example, so below we show the definition for the `ScalarSingletDM` class:

```
SpectrumContents::
  ScalarSingletDM::ScalarSingletDM()
{
   using namespace Par;
   setName("ScalarSingletDM");
   addParameter(mass1,         "vev"      );
   addParameter(dimensionless, "lambda_hS");
   addParameter(Pole_Mass,     "h0");
   addParameter(Pole_Mass,     "S" );
}
```

First, a string name for the spectrum contents should be declared via the `setName` function. After that one simply declares all the parameters that should exist in the `SubSpectrum` wrapper by specifying the tag/string name/indices by which that parameter should be accessed. In the example the index definition argument is omitted, and so the parameter is assumed to be scalar-valued and require

no index. To define an index requirement, one supplies as a final argument a vector of integers, specifying the dimension size of each required index. For example, to define a vector parameter with six entries, and a matrix parameter requiring two indices each of size three, one would write e.g.

```
std::vector<int> v6; v6.push_back(6);
std::vector<int> m3x3; m3x3.push_back(3);
                      m3x3.push_back(3);
addParameter(Pole_Mass, "~d"  , v6);
addParameter(mass2    , "mq2", m3x3);
```

The `SubSpectrumContents` constructor definitions can always be consulted to check exactly what content is required to be retrievable from any wrapper that conforms to it. For reference purposes we provide tables in Appendix C that describe each of the pre-defined `SubSpectrumContents` sets, with cross-referencing to theoretical descriptions of each parameter. The descriptions of pre-defined `SubSpectrum` wrappers in Appendix D also refer to these tables to describe what they contain.

F.2 Wrapping a simple parameter collection

The most basic kind of object for which one may wish to construct an interface is a simple class that contains parameter values. Such an object is far simpler than the typical case that the `SubSpectrum` interface is designed to wrap, and so many of its features are not needed. The functionality provided by wrapping these simple objects in a `SubSpectrum` could be entirely replaced by the standard **GAMBIT** Model system (see Ref. [11]), which already provides a way to deal with simple parameter containers, however, it can be useful to interact with parameters via the Spectrum interface in the cases where they might alternatively be provided by a true spectrum generator. It is also useful to examine this case simply to demonstrate the most basic requirements that a wrapper class must fulfill.

So let us consider the following simple class, and construct a wrapper that links it to the SubSpectrum interface. This wrapper is in fact implemented in **GAMBIT** and provides some scalar-singlet dark matter parameters and Standard Model Higgs sector parameters for use by **DarkBit**; the full in-code implementation can be found in

```
Models/include/gambit/Models/SimpleSpectra/
ScalarSingletDMSimpleSpec.hpp.
```

Example 1 Simple "Model" class for the ScalarSingletDM model

```
namespace Gambit
{
  namespace Models
  {
    struct SingletDMModel
      {
```

```
        double HiggsPoleMass
        double HiggsVEV
        double SingletPoleMass
        double SingletLambda;
    };
  }
}
```

The namespace used here is not important, we specify it just to match the actual code.

To begin wrapping this structure, one must define a specialisation of the `SpecTraits` template class. The purpose of this class is to communicate essential type information to the `SubSpectrum` base classes. Suppose that our wrapper class is to be named `ScalarSingletDMSimpleSpec`; the associated traits class should then be:

```
namespace Gambit
{
 namespace Models
 {
    class ScalarSingletDMSimpleSpec;
 }
 template <>
 struct SpecTraits<
    Models::ScalarSingletDMSimpleSpec>
      : DefaultTraits
 {
  static std::string name()
  { return "ScalarSingletDMSimpleSpec"; }
  typedef
    SpectrumContents::ScalarSingletDM
  Contents;
 };
}
```

Here the namespace *is* important; the original `SpecTraits` template is declared in the `Gambit` namespace, so the specialisation must also live in that namespace. Note that we forward declare the class `ScalarSingletDMSimpleSpec`, which will be our wrapper, because we need it as the template parameter for the traits class.

The required members of this `SpecTraits` specialisation are as follows:

`name()`	A function returning a `std::string` name for the wrapper. This is used in error messages so the class name is generally the logical choice.
`Contents`	A typedef that identifies the contents definition to which this wrapper will conform (see `SubSpectrumContents` in Appendix F.1)

Other `SpecTraits` members can be defined, however, for this simple example we do not need them (the defaults inherited from `DefaultTraits` will suffice). We will return to this in the more complicated examples of Appendices F.3 and F.4.

We are now ready to define the wrapper itself. The declaration is as follows:

Example 2 SubSpectrum wrapper for the ScalarSingletDM model

```
namespace Gambit
{
  namespace Models
  {
    class ScalarSingletDMSimpleSpec
      : public
  Spec<ScalarSingletDMSimpleSpec>
    {
     private:
      SingletDMModel params;

     public:
      ScalarSingletDMSimpleSpec
        (const SingletDMModel& p) :
    params(p) {}

      // Wrapper-side interface functions
      // to SingleDMModel
      double get_HiggsPoleMass()   const;
      double get_HiggsVEV()        const;
      double get_SingletPoleMass() const;
      double get_lambda_hS()       const;

      void set_HiggsPoleMass(double in);
      void set_HiggsVEV(double in);
      void set_SingletPoleMass(double in);
      void set_lambda_hS(double in);

      static GetterMaps fill_getter_maps();
      static SetterMaps fill_setter_maps();
    };
  }
}
```

Let us discuss what is going on here. First, the wrapper class must inherit from the `Spec` class, and provide its own type as the template parameter. This is because we employ the CRTP (curiously recurring template pattern) for static polymorphism, to allow the base class access to the wrapper member functions.

Second, the wrapper contains an instance of the `SingletDMModel` object to which we want to interface. This is not necessary, but it is helpful for maintaining encapsulation.

Next is the constructor. The wrapper writer is quite free to do what they like with this; here we simply use it to initialise the member object.

Following the constructor are a series of "getter" and "setter" functions, which retrieve and set the parameter values we are interested in. These are the functions that we will "hook up" to the `SubSpectrum` "get" and "set" functions. These functions can access the hosted `SingletDMModel` in our example, and so could also perform extra tasks like calling functions of the `SingletDMModel` (if it had any) or performing unit conversions. Our later examples will demonstrate tasks like this. For now, it will suffice for these functions to have the following sort of definition:

```
double get_lambda_hS() const
  { return params.SingletLambda; }

void set_lambda_hS(double in)
  { params.SingletLambda = in; }
```

We will refrain from listing the rest of the definitions in this example because they follow a similar pattern.

Finally, we get to the key part of the wrapper; the `fill_getter_maps` and `fill_setter_maps` functions. These are what define the actual interface to the hosted `Model` (and `Input`) objects. They are simply functions that fill a series of map (i.e. `std::map`) structures in the base class, where these maps are what define which functions are 'hooked up' to the `get`, `set` and `has` functions of the `SubSpectrum` interface (see Sect. 2.3.3). The operation of these 'filler' functions is most easily seen by looking at the definition of one, so let us examine the `fill_getter_maps` definition for our example:

```
static ScalarSingletDMSimpleSpec::GetterMaps
  ScalarSingletDMSimpleSpec::fill_getter_
  maps()
{
  GetterMaps map_collection;
  typedef ScalarSingletDMSimpleSpec Self;

  map_collection[Par::mass1].map0W["vev"]
    = &Self::get_HiggsVEV;
  map_collection[Par::mass1].map0W
  ["lambda_hS"]
    = &Self::get_lambda_hS;
  map_collection[Par::Pole_Mass].map0W["h0"]
    = &Self::get_HiggsPoleMass;
  map_collection[Par::Pole_Mass].map0W["S"]
    = &Self::get_SingletPoleMass;

  return map_collection;
}
```

Here `GetterMaps` is another type inherited from the base `Spec` class, and is the main container object that we use to associate a tag/string pair with a function pointer. So the first entry is what makes it possible for a call `subspectrum.get(Par::mass1,"vev")` to the interface class to in turn call the `get_HiggsVEV` member function of `Model`. The `fill_getter_maps()` function is used by the base `Spec` class to initialise a member variable of type `GetterMaps`, so here we are in fact defining how this member variable will be 'filled'. The `fill_setter_maps` function works in direct analogy to the `fill_getter_maps` function, so we will skip discussion of it.

Note that neither of these filler functions *must* be defined, the wrapper will compile without them and the interface will simply not accept any tag/string pairs to the get/set functions. For example one may fill the 'getter' maps, but not the 'setter' maps, and it will then simply be impossible to change the parameters in the underlying `Model` object via the 'set'

functions, which is not necessarily problematic. However, if one fails to fill the 'getter' maps in accordance with the declared `Contents` of the wrapper, then a runtime error will be thrown as soon as the wrapper object constructor is called, which will present a message telling the user that the wrapper is not correctly defined.

There is a subtlety here; note the `map0W` data member which is accessed from the `map_collection` variable. This member is what defines the function signature of the function pointer to be stored in the `map_collection`. This function signature must be known 'in advance' in order for the base class to call the function pointer, so the function pointer needs to be placed in the correct location within the `map_collection`. In our case the `map0W` data member expects functions with the signature

```
double Self::function(void);
```

that is, they should be a member function of the wrapper, take no arguments, and return a double. The equivalent `map0W` member within `SetterMaps` expects the signature

```
void Self::function(double);
```

that is, functions should be members of the wrapper, should accept a double, and return void.

Functions with a variety of other signatures can also be used, however, they must be placed in the correct data member of `GetterMaps`/`SetterMaps` in order to work. A full list of the allowed function signatures, the way they should be stored in the `GetterMaps`/`SetterMaps`, and the way they can be accessed via the `SubSpectrum` get and set functions, is given in Appendix E. We will see some of these other signatures in use in the more complicated wrapper examples.

F.3 Interfacing directly with member functions of an external class

When one wants to create a `SubSpectrum` wrapper for pre-existing classes, it may be the case that the pre-existing class already has "getter" and "setter" functions, such that one does not need to write new ones in the wrapper. Unfortunately, it is not possible to store functions of arbitrary signature in the `GetterMaps` and `SetterMaps`, and the pre-enabled set of permitted signatures is not very large. More can be added, however it is quite technically involved. If you have a special need to add more, then please contact the authors for advice.

The practical use cases of directly using external member functions is therefore very limited, however, in **SpecBit** we make use of this feature for our most complicated wrapper; the `MSSMSpec` wrapper which interfaces to **FlexibleSUSY** model objects. Furthermore, new auto-generated **Flexible-SUSY** spectrum generators will come with the required set of getters/setters with the correct function signature, so it is

useful to describe this 'shortcut' interface method just for the sake of the **FlexibleSUSY** case.

Suppose that the external class to be wrapped has the following form:

```
struct MyClass
{
 double get_par1() const;
 double get_par2(int i, int j) const;

 void set_par1(double in);
 void set_par2(int i, int j, double in);
};
```

It does not matter where the parameters themselves live at present, we just need the functions that set and retrieve them. In this example we will also show how to deal with functions requiring indices.

These functions can be inserted into the `GetterMaps` and `SetterMaps` of the wrapper via the filler functions as follows:

```
static MyWrapper::GetterMaps
  MyWrapper::fill_getter_maps()
{
 GetterMaps map_collection;
 static const std::set<int> i012 =
   initSet(0,1,2);

 map_collection[tag1].map0["par1"]
   = &Model::get_par1;
 map_collection[tag2].map2["par2"]
   = &FInfo2( &Model::get_par2, i012, i012
   );

 return map_collection;
}
```

and

```
static MyWrapper::SetterMaps
  MyWrapper::fill_setter_maps()
{
 SetterMaps map_collection;
 static const std::set<int> i012 =
   initSet(0,1,2);

 map_collection[tag1].map0["par1"]
   = &Model::set_par1;
 map_collection[tag2].map2["par2"]
   = &FInfo2( &Model::set_par2, i012, i012
   );

 return map_collection;
}
```

where these function signatures match the `map0` and `map2` members of the `GetterMaps` and `SetterMaps` classes, as shown in Appendix E. This appendix also gives an explanation of the `FInfo2` helper class and how it is used to help specify the allowed indices to get and set functions.

In order to call these functions via function pointers, the `SubSpectrum` wrapper needs to know the C++ type of

the hosted class. This can be supplied via a typedef in the `SpecTraits` struct;

```
namespace Gambit
{
 namespace Models {class MyClass;}
 template <>
 struct SpecTraits<MyWrapper> : DefaultTraits
 {
  static std::string name()
  { return "MyWrapper"; }
  typedef SpectrumContents::MyContents
   Contents;
  typedef MyClass Model;
 };
}
```

The wrapper base classes will then be able to use the *MyClass* type via the `Model` typedef. In addition, the wrapper needs to have access to an instance of the *MyClass* type. This must be provided by overloading a special inherited member function `get_Model` in the wrapper, e.g.

```
class MyWrapper
  : public Spec<MyWrapper>
{
 private:
  Model model;

 public:
  MyWrapper(Model& m) : model(m)

  Model& get_Model() { return model; }
  const Model& get_Model() const { return
   model; }

  static GetterMaps fill_getter_maps();
  static SetterMaps fill_setter_maps();
}
```

Note that here `Model` is the typedef for *MyClass*, which is learned via the `SpecTraits` struct. We assume in this example that an instance of `model` is carried as a data member of the wrapper class, which is a good idea for maintaining encapsulation, but it is not strictly required. All that is required is that `get_Model()` return an instance of the `Model` type. Note also that both `const` and non-`const` versions of `get_Model()` are required, because the wrapper base classes need both in order to maintain const-correctness when `const` wrapper objects are used.

F.4 Interfacing with non-class functions

In the previous `SubSpectrum` wrapper examples, we covered interfacing with member functions of wrapper objects, and with member functions of particular external classes. The former can be designed to access external functions or classes in arbitrary ways and so are the most generally useful, while the latter have limited use due to restrictions on the allowed function signature but are convenient for dealing with certain

special cases such as **FlexibleSUSY** classes. Now, we will deal with interfacing to plain functions that are not members of any class.

As with the other permitted kinds of functions, these are also restricted to specific function signatures. In fact, they do not provide any additional functionality over simply using member functions of the wrapper classes, and wrapper member functions are usually a better choice since they have easier access to data members of the wrapper class. However, they can be useful to avoid code repetition if the target functions are to be used in more than one wrapper, for example if they perform some common unit conversions or simple calculations.

The allowed non-class function signatures are described in Appendix E. They may be connected to `GetterMaps` and `SetterMaps` as in the following example:

```cpp
double get_Pole_mElectron(const SMInputs&
    inputs)
 { return inputs.mE; }

static MyWrapper::GetterMaps
  MyWrapper::fill_getter_maps()
{
 GetterMaps map_collection;

 map_collection[Par::Pole_Mass].map0_extraI
 ["e-_1"]
    = &get_Pole_mElectron;

 return map_collection;
}
```

This time, we have used the `map0_extraI` signature, which requires an `Input` object as an argument, and no indices. As with the `Model` object case, the wrapper needs to be informed of the type of the `Input` object, and provided an instance of it, via the `SpecTraits` class and `get_Input` functions, respectively:

```cpp
namespace Gambit
{
 template <>
 struct SpecTraits<MyWrapper> : DefaultTraits
 {
  static std::string name()
  { return "MyWrapper"; }
  typedef SpectrumContents::MyContents
   Contents;
  typedef SMInputs Inputs;
 };
}
```

and

```cpp
class MyWrapper
  : public Spec<MyWrapper>
{
  ...
  Input& get_Input();
  const Input& get_Input() const;
  ...
}
```

Here, `Input` is imagined to be a class containing input information used to setup the wrapper, but which one may also wish to access via the `SubSpectrum` interface along with the rest of the spectrum data. But it is simply an arbitrary class which can be used to pass information to the interface functions.

F.5 Index offsets

In general, `SubSpectrum` wrapper interfaces use a one-based indexing system. For example when retrieving neutralino pole masses from the MSSM spectrum wrappers the lightest neutralino is retrieved with the index 1. However, not all external functions will follow this convention, so it is useful to have a system for converting between index systems. The wrapper system provides this functionality via the `index_offset` wrapper class member function. This function should be overridden in the wrapper in order to define an offset to be added to all index values input by users to the `get` and `set` functions of that wrapper. Its function signature is

```cpp
static int index_offset() {return offset;}
```

So, for example, if *offset* is `-1`, then when a user calls e.g. `subspec.get(Pole_Mass,"~e",1)` the offset will be added to the input index of `1`, resulting in `0`, before being passed to the external function (which will therefore see the index as `0`). If the `index_offset` function is not overridden then a default null offset of `0` will be applied.

Note that this function should have `public` access.

F.6 Interfacing with renormalisation group running functions

Now we have seen all the basic ways of connecting functions of various kinds to the `SubSpectrum` `get`/`set` interface. So far, however, the framework may seem like overkill for the tasks we have performed. Why not just call the external functions and store the return values in the appropriate place? Why store pointers to the functions themselves, or wrappers for those functions? The primary answer here is because we want to be able to call renormalisation group-running functions, and have these affect all the values accessed by the `SubSpectrum` interface. If we simply copied the results of the spectrum calculation into an intermediate object, such as

an `SLHAea` object, then this would not be possible. But with the framework we have set up, when the `get`/`set` functions are called it is possible for them to interact directly with the contents of external spectrum generator codes, and so the values they retrieve can be affected by `RGE` running.

If the external spectrum generator is a C++ code, then connecting the `SubSpectrum` wrapper to the `RGE`-running facilities should be fairly easy. There are two main functions that need to be connected: `GetScale`, and `RunToScaleOverride`. These functions are not required by the `SubSpectrum Contents` definitions, so whether or not a particular `Sub Spectrum` can perform `RGE` running will need to be inferred from the GAMBIT **capability** it is given via whatever module function provides it.

The `GetScale` function is the simplest, and should return the scale at which all running parameters are defined. If there is an analogous function or parameter defined in the host `Model` object then the `GetScale` definition will simply be something like this:

```
double MyWrapper::GetScale() const
{
  return model.get_scale();
}
```

The `RunToScaleOverride` function is wrapped by the user-side `RunToScale` function, which adds additional checks and behaviour modifications. But the `RunToScaleOverride` is what should be overridden in the wrapper and connected to any external `RGE`-running functions. For example:

```
void MyWrapper::RunToScaleOverride(double
    scale)
{
  model.run_to(scale);
}
```

Of course the details for hooking up these functions correctly are entirely dependent on how the external code works. The onus is therefore on the wrapper writer to understand the external code. However, the power of this framework is that once the wrapper is defined, writers of other module functions can interact with spectrum information in a totally generic way, even including `RGE` running.

F.7 Constructing and returning `Spectrum` objects from module functions

So far, we have described how to design a new wrapper class for external spectrum data. However, in GAMBIT, a `Spectrum` object needs to be constructed from the wrapper and returned via a module function before the spectrum data can be used by other module functions. Here we discuss the general requirements of this process.

The details for constructing the wrapper depend on how the wrapper and its constructor are defined, which are up

to the wrapper writer. We will therefore use the `SingletDM` example wrapper from Appendix F.2 to demonstrate the more general requirements for creating and returning the abstract interface. Let us first write out an example module function which does this, and then we will examine it line-by-line.

First, the module function rollcall declaration:

```
#define CAPABILITY SingletDM_spectrum
START_CAPABILITY
  // Create a Spectrum object from SMInputs,
  // SM Higgs parameters, and the SingletDM
  // parameters
  #define FUNCTION get_SingDM_spec
  START_FUNCTION(Spectrum)
  DEPENDENCY(SMINPUTS, SMInputs)
  ALLOW_MODEL_DEPENDENCE(StandardModel_Higgs,
                         SingletDM)
  MODEL_GROUP(higgs,
    (StandardModel_Higgs))
  MODEL_GROUP(singlet, (SingletDM))
  ALLOW_MODEL_COMBINATION(higgs, singlet)
  #undef FUNCTION
#undef CAPABILITY
```

Now the module function itself:

```
void get_SingDM_spec(Spectrum& result)
{
 using namespace Pipes::get_SingDM_spec;
 const SMInputs& sminputs =
  *myPipe::Dep::SMINPUTS;

 Models::SingletDMModel s;
 s.HiggsPoleMass = *Param.at("mH");
 s.HiggsVEV = 1. /
   sqrt(sqrt(2.)*sminputs.GF);
 s.SingletPoleMass = *Param.at("mS");
 s.SingletLambda = *Param.at("lambda_hS");

 Models::
   ScalarSingletDMSimpleSpec
   singletspec(s);

 result =
   Spectrum(singletspec,sminputs,&Param);
}
```

Let us look first at the module function definition, and then afterwards we will examine how the rollcall declaration ensures that the necessary **dependencies** are made available. The first line gives the function signature, which, as always, returns `void` and takes one argument by reference, which is the pre-allocated memory for the storage of the function result. Here we will create a `Spectrum` object so the 'result' type is `Spectrum`. Next, a **dependency** on an `SMInputs` object (as described in Sect. 2.3.5) is retrieved.

Lines 7 to 11 show the construction of an instance of the `SingletDMModel` class, which is the object that we want to wrap. Here we set the parameters contained in the class by extracting them from the `Params` map supplied by GAMBIT via the core hierarchical model system (see Sect. 5 of Ref. [11]). On line 14 we then use the

`SingletDMModel` instance to construct our instance of the `ScalarSingletDMSimpleSpec` wrapper class.

Nearing the end of the module function, we construct on line 16 the full `Spectrum` object out of the components we have prepared, namely the scalar-singlet `SubSpectrum` object and an `SMInputs` object. More generally one can also supply the 'low-energy' `SubSpectrum` component, however, this shortened constructor will automatically generate an `SMSimpleSpec` (see Appendix D) wrapper based on `SMINPUTS` to fulfil this role. The final argument to the `Spectrum` constructor is the set of model parameters which were available to this `Spectrum` when it was constructed, which are retrieved via the `Params` map.

Finally, the newly constructed object is moved into the `result` space of the module function, from where it can be distributed to other module functions by the GAMBIT dependency resolver.

Now, let us return to the module function rollcall declaration and see how it requests the **dependencies** that we require. The first two lines simply declare the name for the **capability** that this function provides. The next two (define/start function) declare the C++ name of the function (which of course must match the function as defined elsewhere). Note here that we also declare the `result` type for this function as `Spectrum`.

Next, we declare the dependency we require. Here it is declared as `SMINPUTS`, with type `SMInputs`. This will then be made available to our module function by the GAMBIT core.

The next few lines declare a joint dependence on model parameters from both `StandardModel_Higgs` and `SingletDM`, that is, they declare that this module function requires parameters from both of these models. For more details on this syntax please see Ref. [11].

With the rollcall declaration done we are finished, and the `Spectrum` object created by the module function `get_SingDM_spec` is made available to the rest of GAMBIT under the **capability** name `SingletDM_spectrum`.

F.8 Scheme-dependence and other special dependency requirements

All of the **capabilities** associated with `Spectrum` types in GAMBIT 1.0.0 are understood to provide running parameters in only one scheme. For example, spectra obtained via the **capability** named `MSSM_spectrum` are understood to provide parameters in the $\overline{DR}$ scheme, while spectra obtained via the **capability** named `SM_spectrum` provide parameters in the $\overline{MS}$ scheme. Thus, when adding new spectrum generators, users should check that they conform to these "defaults" if they write new module functions which promise to provide these existing capabilities (see Tables 1, 2 and 3). If providing a spectrum in a different scheme then a different capability

name should be used, for example `MSSM_spectrum_MSbar`. Of course this spectrum will then not be automatically usable by existing GAMBIT module functions which expect the default $\overline{DR}$ MSSM spectrum, because the new capability will not match with the dependencies of existing functions. Therefore further modification of the dependency hierarchy will generally be necessary to fully integrate such a spectrum with existing module functions. We recommend that users with special requirements of this kind contact the GAMBIT authors for advice on how best to do this.

Along similar lines, some users may wish to receive certain spectrum-related information in a very specific format, for example requiring a running mass extracted at a specific loop order or including specific corrections. Such specificity goes beyond the intended use cases for `Spectrum` objects, so in general this information needs to be recomputed at the point where it is needed from information available via the `Spectrum` wrappers. For example, one might perform some auxiliary calculations in the interface to a backend code that might have a special requirement for some input parameter. If the quantity is required repeatedly then users should consider creating a new **capability**, for example `Higgs_mass_tree_level`, and adding a new module function which performs the necessary calculations to provide this capability.

F.9 Controlling wrapper lifetimes

That covers the construction of a simple wrapper, however, there are a few more subtleties regarding the lifetime (in the sense of time between construction/destruction of the C++ objects as the code runs) of the various wrapper objects to be discussed. There are several constructors for the `Spectrum` object, and it is important to choose the correct one depending on how you want the member `SubSpectrum` objects to be treated. The options are listed below, with the argument accepting the GAMBIT parameter container replaced with `<Param>` for brevity.

`Spectrum()` – Creates an empty object.

`Spectrum(const SubSpectrum& he, const SMInputs& smi, <Params>)` – Constructs a new object, automatically creating an `SMSimpleSpec` as the LE subspectrum, and cloning the "HE" `SubSpectrum` object supplied and taking possession of it.

`Spectrum(const SubSpectrum& le, const SubSpectrum& he, const SMInputs& smi, <Params>)` – Construct new object, cloning both `SubSpectrum` objects supplied and taking possession of them.

`Spectrum(SubSpectrum const le, SubSpectrum const he, const SMInputs& smi, const std)` – Construct new object, wrapping *existing* `SubSpectrum` objects. If the original objects are prematurely destructed

then attempting to access them via the `Spectrum` interface will cause a segmentation fault.

G Worked example of writing a `SubSpectrum` wrapper

In this section we provide a stripped-down tutorial-style example of how to define a new `SubSpectrum` wrapper and return a `Spectrum` object from a module function. Comments and instructions will be kept to a bare minimum, and instead we will simply refer to the appropriate sections of the paper body in which further details can be found.

G.1 FlexibleSUSY MSSM wrapper

The first step in adding a new FlexibleSUSY spectrum generator to SpecBit is to run FlexibleSUSY and generate the C++ code for the new spectrum generator. It then needs to be added to the GAMBIT cmake build system, and the appropriate headers included in certain key SpecBit headers. As mentioned in Sect. 2.1.2 we provide instructions for these steps as separate documentation, which can be found in `gambit/doc/Adding_FlexibleSUSY_Models.txt`, and they will change when future technical improvements to the GAMBIT **BOSS** [11] system permit FlexibleSUSY to be automatically backended. Thus we begin this example by assuming that these steps have been completed. From this point we can begin following the checklist given in Sect. 2.4.

(1) Choose or create a `SubSpectrumContents` definition (Appendix F.1) – The purpose of the `SubSpectrum` classes is to enforce consistency between wrappers which wrap the same essential physics, so one should always try to use an existing definition if possible. However, for the purposes of this example we will pretend that the definition for the MSSM did not previously exist. Thus, in `gambit/Models/include/gambit/Models/ SpectrumContents/ RegisteredSpectra.hpp` we add a line of code:

```
struct MSSM: SubSpectrumContents { MSSM();
   };
```

Next we create a new source file `gambit/Models/ src/ SpectrumContents/MSSM.cpp` with contents which define all the MSSM masses and parameters:

```
#include "gambit/Models/SpectrumContents/
          RegisteredSpectra.hpp"

namespace Gambit
{
  SpectrumContents::MSSM::MSSM()
  {
    setName("MSSM");
    using namespace Par;
```

```
// useful index definitions
std::vector<int> scalar = initVector(1);
std::vector<int> v2      = initVector(2);
std::vector<int> v3      = initVector(3);
std::vector<int> v4      = initVector(4);
std::vector<int> v6      = initVector(6);
std::vector<int> m2x2    = initVector(2,2);
std::vector<int> m3x3    = initVector(3,3);
std::vector<int> m4x4    = initVector(4,4);
std::vector<int> m6x6    = initVector(6,6);
// Spectrum parameters
//            tag,    name,   shape
addParameter(mass2, "BMu" , scalar);
addParameter(mass2, "mHd2", scalar);
addParameter(mass2, "mHu2", scalar);

addParameter(mass2, "mq2", m3x3);
addParameter(mass2, "ml2", m3x3);
addParameter(mass2, "md2", m3x3);
addParameter(mass2, "mu2", m3x3);
addParameter(mass2, "me2", m3x3);

addParameter(mass1, "M1", scalar);
addParameter(mass1, "M2", scalar);
addParameter(mass1, "M3", scalar);
addParameter(mass1, "Mu", scalar);
addParameter(mass1, "vu", scalar);
addParameter(mass1, "vd", scalar);

addParameter(mass1, "TYd", m3x3);
addParameter(mass1, "TYe", m3x3);
addParameter(mass1, "TYu", m3x3);
addParameter(mass1, "ad" , m3x3);
addParameter(mass1, "ae" , m3x3);
addParameter(mass1, "au" , m3x3);

addParameter(dimensionless, "g1", scalar);
addParameter(dimensionless, "g2", scalar);
addParameter(dimensionless, "g3", scalar);

addParameter(dimensionless, "sinW2",
scalar);

addParameter(dimensionless, "Yd", m3x3);
addParameter(dimensionless, "Yu", m3x3);
addParameter(dimensionless, "Ye", m3x3);

addParameter(Pole_Mass, "~g", scalar);
addParameter(Pole_Mass, "~d",   v6);
addParameter(Pole_Mass, "~u",   v6);
addParameter(Pole_Mass, "~e-",  v6);
addParameter(Pole_Mass, "~nu",  v3);
addParameter(Pole_Mass, "~chi+", v2);
addParameter(Pole_Mass, "~chi0", v4);
addParameter(Pole_Mass, "h0",   v2);
addParameter(Pole_Mass, "A0", scalar);
addParameter(Pole_Mass, "H+", scalar);

addParameter(Pole_Mixing, "~d",   m6x6);
addParameter(Pole_Mixing, "~u",   m6x6);
addParameter(Pole_Mixing, "~e-",  m6x6);
addParameter(Pole_Mixing, "~nu",  m3x3);
addParameter(Pole_Mixing, "~chi0", m4x4);
addParameter(Pole_Mixing, "~chi-", m2x2);
addParameter(Pole_Mixing, "~chi+", m2x2);
```

```
      addParameter(Pole_Mixing, "h0",    m2x2);
      addParameter(Pole_Mixing, "A0",    m2x2);
      addParameter(Pole_Mixing, "H+",    m2x2);
   }
}
```

The `SpectrumContents` that we will use are now defined. (2) Write a `SpecTraits` specialisation – In this example, the `getter` and `setter` functions are defined in an external object, so the example in Appendix F.4 is relevant. It is best to define our `SpecTraits` class in the header file for the `SubSpectrum` wrapper class, because the definition needs to be available to the C++ compiler when actual `SubSpectrum` class is defined. We thus begin a new header file `SpecBit/include/gambit/SpecBit/MSSM_FS.hpp` and add the following code (neglecting include guards):

```
#include "gambit/Elements/spec.hpp"
#include "gambit/Models/SpectrumContents/
   RegisteredSpectra.hpp"
#include "gambit/contrib/MassSpectra/
   flexiblesusy/models/MSSM/MSSM_model_slha.hpp"

namespace Gambit
{
  namespace SpecBit
  {
    class MSSM_FS; //Forward declaration
  }

  template <>
  struct SpecTraits<SpecBit::MSSM_FS>
    : DefaultTraits
  {
    static std::string name() {return "MSSM_FS";}
    typedef SpectrumContents::MSSM Contents;
    using namespace flexiblesusy;
    typedef MSSM_slha<Two_scale> Model;
  };
}
```

(3) Write the `SubSpectrum` wrapper class (Appendix F.2) – We can now declare the wrapper class in the same file:

```
namespace Gambit
{
 namespace SpecBit
 {
  class MSSM_FS : public Spec<MSSM_FS>
  {
   private:
    Model fs_model;
    typedef MSSM_FS Self;

   public:
    // Interface function overrides
    static int index_offset() {return -1;}
    double GetScale() const;
    void SetScale(double scale);
    void RunToScaleOverride(double scale);

    //constructors / destructors
```

```
    MSSM_FS() {}
    MSSM_FS(Model& m) : fs_model(m) {}

    // Functions to allow the base 'Spec'
  class
    // access to 'Model'
    Model& get_Model() {return fs_model;}
    const Model& get_Model() const
     {return fs_model;}

    // Functions to fill SLHAea objects
    void add_to_SLHAea(int
  slha_version,SLHAstruct& slha) const;
    SLHAstruct getSLHAea(int getSLHAea)
  const;

    // Map filler overrides
    static GetterMaps fill_getter_maps();
    static SetterMaps fill_setter_maps();

    // Helper functions
    double get_tanbeta();
    double get_DRbar_mA2();
    double get_sinthW2_DRbar();
    double get_MAh1_pole_slha();
    double get_MHpm1_pole_slha();
    void set_MAh1_pole_slha(double);
    void set_MHpm1_pole_slha(double);
    void set_MGluino_pole_slha(double);
    void set_MZ_pole_slha(double);
    void set_MW_pole_slha(double);
    void set_MSu_pole_slha(double, int);
    void set_MSd_pole_slha(double, int);
    void set_MSe_pole_slha(double, int);
    void set_MSv_pole_slha(double, int);
    void set_MCha_pole_slha(double, int);
    void set_MChi_pole_slha(double, int);
    void set_Mhh_pole_slha(double, int);
    void set_ZD_pole_slha(double, int, int);
    void set_ZU_pole_slha(double, int, int);
    void set_ZE_pole_slha(double, int, int);
    void set_ZV_pole_slha(double, int, int);
    void set_ZH_pole_slha(double, int, int);
    void set_ZA_pole_slha(double, int, int);
    void set_ZP_pole_slha(double, int, int);
    void set_ZN_pole_slha(double, int, int);
    void set_UM_pole_slha(double, int, int);
    void set_UP_pole_slha(double, int, int);
    void set_ZH_pole_slha(double, int, int);
  };
 }
}
```

We now begin a new source file

```
    gambit/SpecBit/src/MSSM_FS.cpp
```

to hold the definitions of the wrapper functions:

```
using namespace Gambit;
using namespace SpecBit;

// Fill an SLHAea object
void MSSM_FS::add_to_SLHAea(int
   slha_version,SLHAstruct& slha)
  const
```

```
{
  // Here one needs to extract all the SLHA
  // information from the spectrum object and
  // store it in the supplied SLHAea object.
  // This takes many lines of code, so we
  // exclude it from this example. For a
  // working MSSM example of this function see
  // "gambit/SpecBit/include/gambit/
  // SpecBit/MSSMSpec.hpp".
}

// Return a new (filled) SLHAea object
SLHAstruct MSSM_FS::get_SLHAea(SLHAstruct& slha)
  const
{
  SLHAstruct output;
  add_to_SLHAea(slha_version,output);
  return output;
}

void MSSM_FS::RunToScaleOverride(double scale)
{
  fs_model.run_to(scale);
}

double MSSM_FS::GetScale() const
{
  return fs_model.get_scale();
}

void MSSM_FS::SetScale(double scale)
{
  fs_model.set_scale(scale);
}

// 'extra' function to compute TanBeta
double MSSM_FS::get_tanbeta()
{
  return fs_model.get_vu() / fs_model.get_vd();
}

// 'extra' function to compute mA2
double MSSM_FS::get_DRbar_mA2()
{
  double tb = get_tanbeta();
  double cb = cos(atan(tb));
  double sb = sin(atan(tb));
  return fs_model.get_BMu() / (sb * cb);
}

// 'extra' function to compute DRbar weak
// mixing angle
double MSSM_FS::get_sinthW2_DRbar()
{
  double sthW2 = Utils::sqr(fs_model.get_g1())
   * 0.6 / (0.6 * Utils::sqr(fs_model.get_g1())
   + Utils::sqr(fs_model.get_g2()));
  return sthW2;
}

// Wrapper functions for A0 and H+ getters, to
// retrieve only the non-Goldstone entries.
double MSSM_FS::get_MAh1_pole_slha()
{
  return fs_model.get_MAh_pole_slha(1);
}

double MSSM_FS::get_MHpm1_pole_slha()
{
  return fs_model.get_MHpm_pole_slha(1);
}

// Helper functions for manually setting pole
```

```
// masses. We can use a macro to automate this.
 #define  POLE_MASS_SETTER(NAME) \
void MSSM_FS::set_##NAME##_pole_slha(double mass,\
 int i)\
{\
  fs_model.get_physical_slha().NAME(i) = mass;\
}\

POLE_MASS_SETTER(MSu)
POLE_MASS_SETTER(MSd)
POLE_MASS_SETTER(MSe)
POLE_MASS_SETTER(MSv)
POLE_MASS_SETTER(MCha)
POLE_MASS_SETTER(MChi)
POLE_MASS_SETTER(Mhh)

// Similar for mixings
 #define  POLE_MIXING_SETTER(NAME) \
void MSSM_FS::set_##NAME##_pole_slha(double mass,\
 int i, int j)\
{\
  fs_model.get_physical_slha().NAME(i,j) = mass;\
}\

POLE_MIXING_SETTER(ZD)
POLE_MIXING_SETTER(ZU)
POLE_MIXING_SETTER(ZE)
POLE_MIXING_SETTER(ZV)
POLE_MIXING_SETTER(ZH)
POLE_MIXING_SETTER(ZA)
POLE_MIXING_SETTER(ZP)
POLE_MIXING_SETTER(ZN)
POLE_MIXING_SETTER(UM)
POLE_MIXING_SETTER(UP)
POLE_MIXING_SETTER(ZH)

// Higgs-mass setters
void MSSM_FS::set_MAh1_pole_slha(double mass)
{
  fs_model.get_physical_slha().MAh(1) = mass;
}

void MSSM_FS::set_MHpm1_pole_slha(double mass)
{
  fs_model.get_physical_slha().MHpm(1) = mass;
}

// Pole masses with no indices
 #define  POLE_MASS_SETTER0(FNAME,VNAME)\
void MSSM_FS::set_##FNAME##_pole_slha(double\
 mass)\
{\
  fs_model.get_physical_slha().VNAME = mass;\
}\

POLE_MASS_SETTER0(MGluino,MGlu)
POLE_MASS_SETTER0(MZ,MVZ)
POLE_MASS_SETTER0(MW,MVWm)

// Filler function for getter function maps
MSSM_FS::GetterMaps MSSM_FS::fill_getter_maps()
{
  MSSM_FS::GetterMaps map_collection;

  static const std::set<int> i01 = initSet(0,1);
  static const std::set<int> i012 =
   initSet(0,1,2);
  static const std::set<int> i0123 =
   initSet(0,1,2,3);
  static const std::set<int> i012345 =
   initSet(0,1,2,3,4,5);

  /// mass2 - mass dimension 2 parameters
```

```cpp
// Functions from Model
// (i.e. flexiblesusy::MSSM_slha<Two_scale>)
{
  typename MTget::fmap0 tmp_map;
  tmp_map["BMu"]  = &Model::get_BMu;
  tmp_map["mHd2"] = &Model::get_mHd2;
  tmp_map["mHu2"] = &Model::get_mHu2;
  map_collection[Par::mass2].map0 = tmp_map;
}

// Extra functions from the wrapper
map_collection[Par::mass2].map0W["mA2"] =
 &Self::get_DRbar_mA2;

// Functions from Model with two indices
{
  typename MTget::fmap2 tmp_map;
  tmp_map["mq2"]=FInfo2(&Model::get_mq2,
  i012,        i012);
  tmp_map["ml2"]=FInfo2(&Model::get_ml2,
  i012,       i012);
  tmp_map["md2"]=FInfo2(&Model::get_md2,
  i012,       i012);
  tmp_map["mu2"]=FInfo2(&Model::get_mu2,
  i012,i012);
  tmp_map["me2"]=FInfo2(&Model::get_me2,
  i012,i012);
  map_collection[Par::mass2].map2 = tmp_map;
}

///  mass1 - mass dimension 1 parameters

// Zero-index Member functions of Model
{
  typename MTget::fmap0 tmp_map;
  tmp_map["M1"] = &Model::get_MassB;
  tmp_map["M2"] = &Model::get_MassWB;
  tmp_map["M3"] = &Model::get_MassG;
  tmp_map["Mu"] = &Model::get_Mu;
  tmp_map["vu"] = &Model::get_vu;
  tmp_map["vd"] = &Model::get_vd;
  map_collection[Par::mass1].map0 = tmp_map;
}

// Two-index member functions of Model
{
  typename MTget::fmap2 tmp_map;
  tmp_map["TYd"]=FInfo2(&Model::get_TYd,
  i012,i012);
  tmp_map["TYe"]=FInfo2(&Model::get_TYe,
  i012,i012);
  tmp_map["TYu"]=FInfo2(&Model::get_TYu,
  i012,i012);
  tmp_map["ad"] =FInfo2(&Model::get_TYd,
  i012,i012);
  tmp_map["ae"] =FInfo2(&Model::get_TYe,
  i012,i012);
  tmp_map["au"] =FInfo2(&Model::get_TYu,
  i012,i012);
  map_collection[Par::mass1].map2 = tmp_map;
}

///  dimensionless - mass dimension 0 parameters

// Zero-index member functions of Model
{
  typename MTget::fmap0 tmp_map;
  tmp_map["g1"]= &Model::get_g1;
  tmp_map["g2"]= &Model::get_g2;
  tmp_map["g3"]= &Model::get_g3;
  map_collection[Par::dimensionless].map0 =
   tmp_map;
```

```cpp
}
// Zero-index 'extra' functions from wrapper
{
  typename MTget::fmap0W tmp_map;
  tmp_map["tanbeta"] = &Self::get_tanbeta;
  tmp_map["sinW2"]   = &Self::get_sinthW2_DRbar;
  map_collection[Par::dimensionless].map0W
  =tmp_map;
}

// Two-index member functions of Model
{
  typename MTget::fmap2 tmp_map;
  tmp_map["Yd"]= FInfo2(&Model::get_Yd,i012,
   i012);
  tmp_map["Yu"]= FInfo2(&Model::get_Yu,i012,
   i012);
  tmp_map["Ye"]= FInfo2(&Model::get_Ye,i012,
   i012);
  map_collection[Par::dimensionless].map2 =
   tmp_map;
}

/// Pole_Mass - Pole mass parameters

// Zero-index member functions of Model
{
  typename MTget::fmap0 tmp_map;
  tmp_map["W+"] = &Model::get_MVWm_pole_slha;
  tmp_map["~g"] = &Model::get_MGlu_pole_slha;
  map_collection[Par::Pole_Mass].map0 = tmp_map;
}

// Zero-index 'extra' functions from wrapper
{
  typename MTget::fmap0W tmp_map;
  tmp_map["A0"] = &Self::get_MAh1_pole_slha;
  tmp_map["H+"] = &Self::get_MHpm1_pole_slha;
  map_collection[Par::Pole_Mass].map0W =
   tmp_map;
}

// One-index member functions of Model
{
  typename MTget::fmap1 tmp_map;
  tmp_map["~d"] = FInfo1(
   &Model::get_MSd_pole_slha, i012345 );
  tmp_map["~u"] = FInfo1(
   &Model::get_MSu_pole_slha, i012345 );
  tmp_map["~e-"]= FInfo1(
   &Model::get_MSe_pole_slha, i012345 );
  tmp_map["~nu"]= FInfo1(
   &Model::get_MSv_pole_slha, i012 );
  tmp_map["h0"] = FInfo1(
   &Model::get_Mhh_pole_slha, i01 );
  tmp_map["~chi+"] = FInfo1(
   &Model::get_MCha_pole_slha, i01 );
  tmp_map["~chi0"] = FInfo1(
   &Model::get_MChi_pole_slha, i0123 );
  map_collection[Par::Pole_Mass].map1 = tmp_map;
}

/// Pole_Mixing - Pole mass parameters

// Two-index member functions of Model
{
  typename MTget::fmap2 tmp_map;
  tmp_map["~d"] = FInfo2(
   &Model::get_ZD_pole_slha, i012345, i012345);
  tmp_map["~nu"]= FInfo2(
   &Model::get_ZV_pole_slha, i012, i012);
  tmp_map["~u"] = FInfo2(
```

```
        &Model::get_ZU_pole_slha, i012345, i012345);
    tmp_map["~e-"]= FInfo2(
        &Model::get_ZE_pole_slha, i012345, i012345);
    tmp_map["h0"] = FInfo2(
        &Model::get_ZH_pole_slha, i01, i01);
    tmp_map["A0"] = FInfo2(
        &Model::get_ZA_pole_slha, i01, i01);
    tmp_map["H+"] = FInfo2(
        &Model::get_ZP_pole_slha, i01, i01);
    tmp_map["~chi0"] = FInfo2(
        &Model::get_ZN_pole_slha, i0123, i0123);
    tmp_map["~chi-"] = FInfo2(
        &Model::get_UM_pole_slha, i01, i01);
    tmp_map["~chi+"] = FInfo2(
        &Model::get_UP_pole_slha, i01, i01);
    map_collection[Par::Pole_Mixing].map2 =
        tmp_map;
  }

  return map_collection;
}

// Filler function for setter function maps
MSSM_FS::SetterMaps MSSMSpec_FS::
 fill_setter_maps()
{
  MSSM_FS::SetterMaps map_collection;

  static const std::set<int> i01 = initSet(0,1);
  static const std::set<int> i012 =
   initSet(0,1,2);
  static const std::set<int> i0123 =
   initSet(0,1,2,3);
  static const std::set<int> i012345 =
   initSet(0,1,2,3,4,5);

  /// mass2 - mass dimension 2 parameters

  // Zero-index member functions of Model
  {
    typename MTset::fmap0 tmp_map;
    tmp_map["BMu"]  = &Model::set_BMu;
    tmp_map["mHd2"] = &Model::set_mHd2;
    tmp_map["mHu2"] = &Model::set_mHu2;
    map_collection[Par::mass2].map0 = tmp_map;
  }

  // Two-index member functions of Model
  {
    typename MTset::fmap2 tmp_map;
    tmp_map["mq2"]=FInfo2(&Model::set_mq2,
    i012,i012);
    tmp_map["ml2"]=FInfo2(&Model::set_ml2,
    i012,i012);
    tmp_map["md2"]=FInfo2(&Model::set_md2,
    i012,i012);
    tmp_map["mu2"]=FInfo2(&Model::set_mu2,
    i012,i012);
    tmp_map["me2"]=FInfo2(&Model::set_me2,
    i012,i012);
    map_collection[Par::mass2].map2 = tmp_map;
  }

  /// mass1 - mass dimension 1 parameters

  // Zero-index member functions of Model
  {
    typename MTset::fmap0 tmp_map;
    tmp_map["M1"]= &Model::set_MassB;
    tmp_map["M2"]= &Model::set_MassWB;
    tmp_map["M3"]= &Model::set_MassG;
    tmp_map["Mu"]= &Model::set_Mu;
    tmp_map["vu"]= &Model::set_vu;
```

```
    tmp_map["vd"]= &Model::set_vd;
    map_collection[Par::mass1].map0 =
     tmp_map;
  }

// Two-index member functions of model object
  {
    typename MTset::fmap2 tmp_map;
    tmp_map["TYd"]=FInfo2(&Model::set_TYd,
    i012,i012);
    tmp_map["TYe"]=FInfo2(&Model::set_TYe,
    i012,i012);
    tmp_map["TYu"]=FInfo2(&Model::set_TYu,
    i012,i012);
    tmp_map["ad"] =FInfo2(&Model::set_TYd,i012,
     i012);
    tmp_map["ae"] =FInfo2(&Model::set_TYe,i012,
     i012);
    tmp_map["au"] =FInfo2(&Model::set_TYu,i012,
     i012);
    map_collection[Par::mass1].map2 = tmp_map;
  }

/// dimensionless - mass dimension 0 parameters
//
// Zero-index member functions of Model
  {
    typename MTset::fmap0 tmp_map;
    tmp_map["g1"]= &Model::set_g1;
    tmp_map["g2"]= &Model::set_g2;
    tmp_map["g3"]= &Model::set_g3;
    map_collection[Par::dimensionless].map0 =
     tmp_map;
  }

// Two-index member functions of Model
  {
    typename MTset::fmap2 tmp_map;
    tmp_map["Yd"]= FInfo2(&Model::set_Yd,i012,
     i012);
    tmp_map["Yu"]= FInfo2(&Model::set_Yu,i012,
     i012);
    tmp_map["Ye"]= FInfo2(&Model::set_Ye,i012,
     i012);
    map_collection[Par::dimensionless].map2 =
     tmp_map;
  }

/// Pole_Mass parameters

// Zero-index 'extra' functions from wrapper
  {
    typename MTset::fmap0W tmp_map;
    tmp_map["~g"] = &Self::set_MGluino_pole_slha;
    tmp_map["A0"] = &Self::set_MAh1_pole_slha;
    tmp_map["H+"] = &Self::set_MHpm1_pole_slha;
    tmp_map["W+"] = &Self::set_MW_pole_slha;
    map_collection[Par::Pole_Mass].map0W =
     tmp_map;
  }

// One-index 'extra' functions from wrapper
  {
    typename MTset::fmap1W tmp_map;
    tmp_map["~u"] = FInfo1W(
     &Self::set_MSu_pole_slha, i012345 );
    tmp_map["~d"] = FInfo1W(
     &Self::set_MSd_pole_slha, i012345 );
    tmp_map["~e-"]= FInfo1W(
     &Self::set_MSe_pole_slha, i012345 );
    tmp_map["~nu"]= FInfo1W(
     &Self::set_MSv_pole_slha, i012 );
    tmp_map["~chi+"] = FInfo1W(
```

```
    &Self::set_MCha_pole_slha, i01 );
  tmp_map["~chi0"] = FInfo1W(
    &Self::set_MChi_pole_slha, i0123 );
  tmp_map["h0"] = FInfo1W(
    &Self::set_Mhh_pole_slha, i01 );
  map_collection[Par::Pole_Mass].map1W =
    tmp_map;
}

/// Pole_Mixing parameters

// Two-index 'extra' functions from wrapper
{
  typename MTset::fmap2_extraM tmp_map;
  tmp_map["~d"] = FInfo2W(
   &Self::set_ZD_pole_slha, i012345, i012345);
  tmp_map["~nu"]= FInfo2W(
   &Self::set_ZV_pole_slha, i012, i012);
  tmp_map["~u"] = FInfo2W(
   &Self::set_ZU_pole_slha, i012345, i012345);
  tmp_map["~e-"]= FInfo2W(
   &Self::set_ZE_pole_slha, i012345, i012345);
  tmp_map["h0"] = FInfo2W(
   &Self::set_ZH_pole_slha, i01, i01);
  tmp_map["A0"] = FInfo2W(
   &Self::set_ZA_pole_slha, i01, i01);
  tmp_map["H+"] = FInfo2W(
   &Self::set_ZP_pole_slha, i01, i01);
  tmp_map["~chi0"] = FInfo2W(
   &Self::set_ZN_pole_slha, i0123, i0123);
  tmp_map["~chi-"] = FInfo2W(
   &Self::set_UM_pole_slha, i01, i01);
  tmp_map["~chi+"] = FInfo2W(
   &Self::set_UP_pole_slha, i01, i01);
  map_collection[Par::Pole_Mixing].map2W =
    tmp_map;
}

  return map_collection;
}
```

(4) Write a module function to construct and return a `Spectrum` interface object connected to the new wrapper (Appendix F.7) – Let us suppose that the new module function will be added to SpecBit. Module functions that construct spectrum objects for the MSSM are declared in the header `SpecBit/include/gambit/SpecBit/SpecBit_MSSM_rollcall.hpp`. To this file, within the **capability** block beginning `#define CAPABILITY unimproved_MSSM _spectrum`, we add the following declaration:

```
#define FUNCTION get_MSSM_FS_spectrum
START_FUNCTION(Spectrum)
ALLOW_MODELS(MSSM63atQ)
DEPENDENCY(SMINPUTS, SMInputs)
#undef FUNCTION
```

We assume here that the new FlexibleSUSY spectrum generator has been defined such that the soft masses are input at a user-specified scale Q, such that the matching GAMBIT model parameters are `MSSM63atQ`. We therefore declare `ALLOW_MODELS(MSSM63atQ)`, so that we will be able to access these parameters in our module function. We will also need SM SLHA2 parameters, but it is more convenient to obtain them via an `SMInputs` object rather than directly from GAMBIT model parameters, so we declare a **depen-**

dency on `SMINPUTS`, which can be provided by an existing SpecBit module function.

Next we add the module function definition to an appropriate source file; `gambit/SpecBit/ src/SpecBit_MSSM.cpp` is suitable for this example:

```
// Runs MSSM spectrum generator with
// input at scale Q
void get_MSSM_FS_spectrum (Spectrum& result)
{
 namespace myPipe =
   Pipes::get_MSSM_FS_spectrum;
 namespace ss = softsusy;
 const SMInputs& sminputs =
  *myPipe::Dep::SMINPUTS;
 flexiblesusy::MSSM_input_parameters input;

 // Transfer input parameters to FlexibleSUSY
 input.Qin = *myPipe::Param.at("Qin");

 //double valued parameters
 input.TanBeta     = *Param.at("TanBeta");
 input.SignMu      = *Param.at("SignMu");
 input.mHu2IN      = *Param.at("mHu2");
 input.mHd2IN      = *Param.at("mHd2");
 input.MassBInput  = *Param.at("M1");
 input.MassWBInput = *Param.at("M2");
 input.MassGInput  = *Param.at("M3");

 //3x3 matrices; filled with the help of
 // a convenience function
 #define  FILL_3X3_SYM \
  fill_3x3_symmetric_parameter_matrix
 #define  FILL_3X3 \
  fill_3x3_parameter_matrix
 input.mq2Input = FILL_3X3_SYM("mq2", Param);
 input.ml2Input = FILL_3X3_SYM("ml2", Param);
 input.md2Input = FILL_3X3_SYM("md2", Param);
 input.mu2Input = FILL_3X3_SYM("mu2", Param);
 input.me2Input = FILL_3X3_SYM("me2", Param);
 input.Aeij = FILL_3X3("Ae", Param);
 input.Adij = FILL_3X3("Ad", Param);
 input.Auij = FILL_3X3("Au", Param);

 // Construct spectrum generator input objects

 // SoftSUSY object used to set quark and
   lepton
 // masses and gauge couplings in QEDxQCD
 // effective theory.
 softsusy::QedQcd oneset;

 // Fill QedQcd object with SMInputs values
 oneset.setPoleMt(sminputs.mT);
 oneset.setPoleMtau(sminputs.mTau);
 oneset.setMbMb(sminputs.mBmB);
 oneset.setMass(ss::mDown, sminputs.mD);
 oneset.setMass(ss::mUp,   sminputs.mU);
 oneset.setMass(ss::mStrange, sminputs.mS);
 oneset.setMass(ss::mCharm,   sminputs.mCmC);
 oneset.setAlpha(ss::ALPHA,
   1./sminputs.alphainv);
 oneset.setAlpha(ss::ALPHAS, sminputs.alphaS);
 oneset.setMass(ss::mElectron, sminputs.mE);
```

```cpp
oneset.setMass(ss::mMuon,      sminputs.mMu);
oneset.setPoleMZ(sminputs.mZ);

// Run everything to Mz
oneset.toMz();

// Create spectrum generator object
flexiblesusy::MSSM_spectrum_generator
<Two_scale> spectrum_generator;

// Generate spectrum
spectrum_generator.run(oneset, input);

// Create SubSpectrum wrapper objects
MSSM_FS
  mssmspec(spectrum_generator.get_model())
QedQcdWrapper qedqcdspec(oneset,sminputs);

// Check for problems during spectrum
  generation
if( spectrum_generator.get_problems()
    .have_problem() )
{
  std::ostringstream msg;
  problems.print_problems(msg);
  invalid_point().raise(msg.str());
}

// Package QedQcd SubSpectrum object, MSSM
// SubSpectrum object, and SMInputs struct
// into a 'full' Spectrum object
result = Spectrum(qedqcdspec,mssmspec,
                 sminputs,myPipe::Param);
}
```

The example is now complete. Various aspects related to error checking, code re-use between module functions, and altering the behaviour of the spectrum generator via YAML options, have been removed in order to keep the example simpler. It is also useful to write several 'auxiliary' module functions to perform helper tasks on the new spectrum, for instance transforming the **capability**, extracting `SLHAea` objects, and similar tasks. See the module functions in `"gambit/SpecBit/src/SpecBit_MSSM.cpp"` for examples of these.

7 Glossary

Here we explain some terms that have specific technical definitions in GAMBIT.

backend An external code containing useful functions (or variables) that one might wish to call (or read/write) from a **module function**.

backend function A function contained in a **backend**. It calculates a specific quantity indicated by its **capability**. Its capability and call signature are defined in the backend's **frontend header**.

backend requirement A declaration that a given **module function** needs to be able to call a **backend function** or use a **backend variable**, identified according to its **capability** and type(s). Backend requirements are declared in module functions' entries in **rollcall headers**.

backend variable A global variable contained in a **backend**. It corresponds to a specific quantity indicated by its **capability**. Its capability and type are defined in the backend's **frontend header**.

BOSS The Backend-On-a-Stick script, used for pre-processing C++ backend code to allow GAMBIT to dynamically load classes from it.

capability A name describing the actual quantity that is calculated by a module or backend function. This is one possible place for units to be noted; the other is in the documented description of the capability (see Sect. 10.7 of Ref. [11]).

dependency A declaration that a given **module function** needs to be able to access the result of another module function, identified according to its **capability** and type. Dependencies are declared in module functions' entries in **rollcall headers**.

dependency resolver The component of the GAMBIT Core that performs **dependency resolution**.

dependency resolution The process by which GAMBIT determines the **module functions**, **backend functions** and **backend variables** needed and allowed for a given scan, connects them to each others' **dependencies** and **backend requirements**, and determines the order in which they must be called.

dependency tree A result of **dependency resolution**; a directed acyclic graph of **module functions** connected by resolved **dependencies**. See Fig. 5 of Ref. [11] for an example.

frontend The interface between GAMBIT and a given **backend**, consisting of a **frontend header** plus optional source files and type headers.

frontend header The C++ header in which the **frontend** to a given **backend** is declared.

module A subset of GAMBIT functions following a common theme, able to be compiled into a standalone library. Although **module** often gets used as shorthand for **physics module**, this term technically also includes the GAMBIT scanning module ScannerBit.

module function A function contained in a **physics module**. It calculates a specific quantity indicated by its **capability** and **type**, as declared in the module's **rollcall header**. It takes only one argument, by reference (the quantity to be calculated), and has a void return type.

physics module Any **module** other than ScannerBit, containing a collection of **module functions** following a common physics theme.

rollcall header The C++ header in which a given **physics module** and its **module functions** are declared.

type A general fundamental or derived C++ type, often referring to the type of the **capability** of a **module function**.

References

1. P.Z. Skands et al., SUSY Les Houches accord: interfacing SUSY spectrum calculators, decay packages, and event generators. JHEP **07**, 036 (2004). arXiv:hep-ph/0311123

2. B.C. Allanach et al., SUSY Les Houches accord 2. Comput. Phys. Commun. **180**, 8–25 (2009). arXiv:0801.0045

3. A. Djouadi, M.M. Mühlleitner, M. Spira, Decays of supersymmetric particles: the program SUSY-HIT (SUspect-SdecaY-Hdecay-InTerface). Acta Phys. Pol. **38**, 635–644 (2007). arXiv:hep-ph/0609292

4. U. Ellwanger, J.F. Gunion, C. Hugonie, NMHDECAY: a Fortran code for the Higgs masses, couplings and decay widths in the NMSSM. JHEP **02**, 066 (2005). arXiv:hep-ph/0406215

5. U. Ellwanger, C. Hugonie, NMHDECAY 2.0: an updated program for sparticle masses, Higgs masses, couplings and decay widths in the NMSSM. Comput. Phys. Commun. **175**, 290–303 (2006). arXiv:hep-ph/0508022

6. U. Ellwanger, C. Hugonie, NMSPEC: a Fortran code for the sparticle and Higgs masses in the NMSSM with GUT scale boundary conditions. Comput. Phys. Commun. **177**, 399–407 (2007). arXiv:hep-ph/0612134

7. B.C. Allanach, S. Kraml, W. Porod, Theoretical uncertainties in sparticle mass predictions from computational tools. JHEP **03**, 016 (2003). arXiv:hep-ph/0302102

8. B.C. Allanach, A. Djouadi, J.L. Kneur, W. Porod, P. Slavich, Precise determination of the neutral Higgs boson masses in the MSSM. JHEP **09**, 044 (2004). arXiv:hep-ph/0406166

9. F. Staub, P. Athron, U. Ellwanger, R. Gröber, M. Mühlleitner, P. Slavich, A. Voigt, Higgs mass predictions of public NMSSM spectrum generators. Comput. Phys. Commun. **202**, 113–130 (2016). https://doi.org/10.1016/j.cpc.2016.01.005

10. P. Drechsel, R. Gröber, S. Heinemeyer, M.M. Muhlleitner, H. Rzehak, G.Weiglein, Higgs-boson masses and mixing matrices in the NMSSM: analysis of on-shell calculations. Eur. Phys. J. **C77**(6), 366 (2017). https://doi.org/10.1140/epjc/s10052-017-4932-4

11. The GAMBIT Collaboration: P. Athron, et al., GAMBIT: the global and modular beyond-the-standard-model inference tool. Eur. Phys. J. **C77**(11), 784 (2017). https://doi.org/10.1140/epjc/s10052-017-5321-8

12. The GAMBIT Scanner Workgroup: G.D. Martinez, J. McKay, B. Farmer, P. Scott, E. Roebber, A. Putze, J. Conrad, Comparison of statistical sampling methods with ScannerBit, the GAMBIT scanning module. Eur. Phys. J. **C77**(11), 761 (2017). https://doi.org/10.1140/epjc/s10052-017-5274-y

13. GAMBIT Dark Matter Workgroup: T. Bringmann et al., DarkBit: A GAMBIT module for computing dark matter observables and likelihoods. Eur. Phys. J. **C77**(12), 831 (2017). https://doi.org/10.1140/epjc/s10052-017-5155-4

14. The GAMBIT Scanner Workgroup: C. Balázs et al., ColliderBit: a GAMBIT module for the calculation of high-energy collider observables and likelihoods. GAMBIT collaboration. Eur. Phys. J. **C77**(11), 795 (2017). https://doi.org/10.1140/epjc/s10052-017-5285-8

15. The GAMBIT Flavour Workgroup collaboration: F.U. Bernlochner et al., FlavBit: A GAMBIT module for computing flavour observables and likelihoods. Eur. Phys. J. **C77**(11), 786 (2017). https://doi.org/10.1140/epjc/s10052-017-5157-2

16. The GAMBIT Collaboration: P. Athron et al., Global fits of GUT-scale SUSY models with GAMBIT. Eur. Phys. J. **C77**(12), 824 (2017). https://doi.org/10.1140/epjc/s10052-017-5167-0

17. GAMBIT Collaboration: P. Athron, C. Balázs et al., A global fit of the MSSM with GAMBIT. Eur. Phys. J. C (2017, under final review). arXiv:1705.07917

18. GAMBIT Collaboration: P. Athron, C. Balázs et al., Status of the scalar singlet dark matter model. Eur. Phys. J. C **77**, 568 (2017). arXiv:1705.07931

19. P. Athron, J.-H. Park, D. Stöckinger, A. Voigt, FlexibleSUSY—a spectrum generator for supersymmetric models. Comput. Phys. Commun. **190**, 139–172 (2015). arXiv:1406.2319

20. W. Porod, SPheno, a program for calculating supersymmetric spectra, SUSY particle decays and SUSY particle production at e^+e^- colliders. Comput. Phys. Commun. **153**, 275–315 (2003). arXiv:hep-ph/0301101

21. W. Porod, F. Staub, SPheno 3.1: extensions including flavour, CP-phases and models beyond the MSSM. Comput. Phys. Commun. **183**, 2458–2469 (2012). arXiv:1104.1573

22. S. Heinemeyer, W. Hollik, G. Weiglein, FeynHiggs: a program for the calculation of the masses of the neutral CP even Higgs bosons in the MSSM. Comput. Phys. Commun. **124**, 76–89 (2000). arXiv:hep-ph/9812320

23. S. Heinemeyer, W. Hollik, G. Weiglein, The masses of the neutral CP—even Higgs bosons in the MSSM: accurate analysis at the two loop level. Eur. Phys. J. C **9**, 343–366 (1999). arXiv:hep-ph/9812472

24. G. Degrassi, S. Heinemeyer, W. Hollik, P. Slavich, G. Weiglein, Towards high precision predictions for the MSSM Higgs sector. Eur. Phys. J. C **28**, 133–143 (2003). arXiv:hep-ph/0212020

25. M. Frank, T. Hahn et al., The Higgs boson masses and mixings of the complex MSSM in the Feynman-diagrammatic approach. JHEP **02**, 047 (2007). arXiv:hep-ph/0611326

26. T. Hahn, S. Heinemeyer, W. Hollik, H. Rzehak, G. Weiglein, High-precision predictions for the light CP-even Higgs boson mass of the minimal supersymmetric standard model. Phys. Rev. Lett. **112**, 141801 (2014). arXiv:1312.4937

27. H. Bahl, W. Hollik, Precise prediction for the light MSSM Higgs boson mass combining effective field theory and fixed-order calculations. Eur. Phys. J. C **76**, 499 (2016). arXiv:1608.01880

28. A. Djouadi, J. Kalinowski, M. Spira, HDECAY: a program for Higgs boson decays in the standard model and its supersymmetric extension. Comput. Phys. Commun. **108**, 56–74 (1998). arXiv:hep-ph/9704448

29. M. Spira, QCD effects in Higgs physics. Fortschr. Phys. **46**, 203–284 (1998). arXiv:hep-ph/9705337

30. J.M. Butterworth et al., The Tools and Monte Carlo Working Group, in *Summary Report from the Les Houches 2009 Workshop on TeV Colliders, in Physics at TeV colliders. Proceedings, 6th Workshop, Dedicated to Thomas Binoth, Les Houches, France, June 8–26, 2009* (2010). arXiv:1003.1643

31. M. Muhlleitner, A. Djouadi, Y. Mambrini, SDECAY: a Fortran code for the decays of the supersymmetric particles in the MSSM. Comput. Phys. Commun. **168**, 46–70 (2005). arXiv:hep-ph/0311167

32. F. Mahmoudi, SuperIso: a program for calculating the isospin asymmetry of $B \rightarrow K^*\gamma$ in the MSSM. Comput. Phys. Commun. **178**, 745 (2008). arXiv:0710.2067

33. F. Mahmoudi, SuperIso v2.3: a program for calculating flavor physics observables in supersymmetry. Comput. Phys. Commun. **180**, 1579 (2009). arXiv:0808.3144

34. F. Mahmoudi, SuperIso v3.0, flavor physics observables calculations: extension to NMSSM. Comput. Phys. Commun. **180**, 1718 (2009)

35. P. Athron, M. Bach et al., GM2Calc: precise MSSM prediction for (g-2) of the muon. Eur. Phys. J. C **76**, 62 (2016). arXiv:1510.08071

36. B.C. Allanach, SOFTSUSY: a program for calculating supersymmetric spectra. Comput. Phys. Commun. **143**, 305–331 (2002). arXiv:hep-ph/0104145

37. B.C. Allanach, M.A. Bernhardt, Including R-parity violation in the numerical computation of the spectrum of the minimal supersymmetric standard model: SOFTSUSY. Comput. Phys. Commun. **181**, 232–245 (2010). arXiv:0903.1805

38. B.C. Allanach, C.H. Kom, M. Hanussek, Computation of neutrino masses in R-parity violating supersymmetry: SOFTSUSY3.2. Comput. Phys. Commun. **183**, 785–793 (2012). arXiv:1109.3735

39. B.C. Allanach, A. Bednyakov, R. Ruiz de Austri, Higher order corrections and unification in the minimal supersymmetric standard model: SOFTSUSY3.5. Comput. Phys. Commun. **189**, 192–206 (2015). arXiv:1407.6130

40. A. Djouadi, J.-L. Kneur, G. Moultaka, SuSpect: a Fortran code for the supersymmetric and Higgs particle spectrum in the MSSM. Comput. Phys. Commun. **176**, 426–455 (2007). arXiv:hep-ph/0211331

41. B.C. Allanach, P. Athron, L.C. Tunstall, A. Voigt, A.G. Williams, Next-to-minimal SOFTSUSY. Comput. Phys. Commun. **185**, 2322–2339 (2014). arXiv:1311.7659

42. K. Ender, T. Graf, M. Muhlleitner, H. Rzehak, Analysis of the NMSSM Higgs boson masses at one-loop level. Phys. Rev. D **85**, 075024 (2012). arXiv:1111.4952

43. T. Graf, R. Grober, M. Muhlleitner, H. Rzehak, K. Walz, Higgs boson masses in the complex NMSSM at one-loop level. JHEP **10**, 122 (2012). arXiv:1206.6806

44. J. Baglio, R. Gröber et al., NMSSMCALC: a program package for the calculation of loop-corrected Higgs boson masses and decay widths in the (complex) NMSSM. Comput. Phys. Commun. **185**, 3372–3391 (2014). arXiv:1312.4788

45. S.F. King, M. Muhlleitner, R. Nevzorov, K. Walz, Exploring the CP-violating NMSSM: EDM constraints and phenomenology. Nucl. Phys. B **901**, 526–555 (2015). arXiv:1508.03255

46. F. Staub, SARAH. arXiv:0806.0538

47. F. Staub, Automatic calculation of supersymmetric renormalization group equations and self energies. Comput. Phys. Commun. **182**, 808–833 (2011). arXiv:1002.0840

48. F. Staub, SARAH 3.2: Dirac Gauginos, UFO output, and more. Comput. Phys. Commun. **184**, 1792–1809 (2013). arXiv:1207.0906

49. F. Staub, SARAH 4: a tool for (not only SUSY) model builders. Comput. Phys. Commun. **185**, 1773–1790 (2014). arXiv:1309.7223

50. M.D. Goodsell, K. Nickel, F. Staub, Two-loop Higgs mass calculations in supersymmetric models beyond the MSSM with SARAH and SPheno. Eur. Phys. J. C **75**, 32 (2015). arXiv:1411.0675

51. W. Frisch, H. Eberl, H. Hlucha, HFOLD—a program package for calculating two-body MSSM Higgs decays at full one-loop level. Comput. Phys. Commun. **182**, 2219–2226 (2011). arXiv:1012.5025

52. H. Hlucha, H. Eberl, W. Frisch, SFOLD—a program package for calculating two-body sfermion decays at full one-loop level in the MSSM. Comput. Phys. Commun. **183**, 2307–2312 (2012). arXiv:1104.2151

53. J. Pardo Vega, G. Villadoro, SusyHD: Higgs mass determination in supersymmetry. JHEP **07**, 159 (2015). arXiv:1504.05200

54. E. Bagnaschi, F. Brümmer, W. Buchmüller, A. Voigt, G. Weiglein, Vacuum stability and supersymmetry at high scales with two Higgs doublets. JHEP **03**, 158 (2016). arXiv:1512.07761

55. P. Athron, J.-H. Park, T. Steudtner, D. Stöckinger, A. Voigt, Precise Higgs mass calculations in (non-)minimal supersymmetry at both high and low scales. JHEP **01**, 079 (2017). arXiv:1609.00371

56. M. Cacciari, G.P. Salam, G. Soyez, FastJet user manual. Eur. Phys. J. C **72**, 1896 (2012). arXiv:1111.6097

57. W. Siegel, Supersymmetric dimensional regularization via dimensional reduction. Phys. Lett. B **84**, 193–196 (1979)

58. D.M. Capper, D.R.T. Jones, P. van Nieuwenhuizen, Regularization by dimensional reduction of supersymmetric and nonsupersymmetric gauge theories. Nucl. Phys. B **167**, 479–499 (1980)

59. I. Jack, D.R.T. Jones, S.P. Martin, M.T. Vaughn, Y. Yamada, Decoupling of the epsilon scalar mass in softly broken supersymmetry. Phys. Rev. D **50**, R5481–R5483 (1994). arXiv:hep-ph/9407291

60. W.A. Bardeen, A.J. Buras, D.W. Duke, T. Muta, Deep inelastic scattering beyond the leading order in asymptotically free gauge theories. Phys. Rev. D **18**, 3998 (1978)

61. Particle Data Group: K.A. Olive et al., Review of particle physics. Chin. Phys. C **38**, 090001 (2014)

62. G. Bélanger, K. Kannike, A. Pukhov, M. Raidal, Z_3 scalar singlet dark matter. JCAP **1**, 022 (2013). arXiv:1211.1014

63. T. Alanne, K. Tuominen, V. Vaskonen, Strong phase transition, dark matter and vacuum stability from simple hidden sectors. Nucl. Phys. B **889**, 692–711 (2014). arXiv:1407.0688

64. N. Khan, S. Rakshit, Study of electroweak vacuum metastability with a singlet scalar dark matter. Phys. Rev. D **90**, 113008 (2014). arXiv:1407.6015

65. J.A. Aguilar-Saavedra et al., Supersymmetry parameter analysis: SPA convention and project. Eur. Phys. J. C **46**, 43–60 (2006). arXiv:hep-ph/0511344

66. F. Staub, W. Porod, Improved predictions for intermediate and heavy Supersymmetry in the MSSM and beyond. Eur. Phys. J. **C77**(5), 338 (2017). https://doi.org/10.1140/epjc/s10052-017-4893-7

67. M. Sher, Electroweak Higgs potential and vacuum stability. Phys. Rep. **179**, 273 (1989)

68. J. Elias-Miró, J.R. Espinosa et al., Higgs mass implications on the stability of the electroweak vacuum. Phys. Lett. Sect. B Nucl. Elementary Part. High Energy Phys. **709**, 222–228 (2012). arXiv:1112.3022

69. S. Alekhin, A. Djouadi, S. Moch, The top quark and Higgs boson masses and the stability of the electroweak vacuum. Phys. Lett. Sect. B Nucl. Elementary Part. High Energy Phys. **716**, 214–219 (2012). arXiv:1207.0980

70. F. Bezrukov, M.Yu. Kalmykov, B.A. Kniehl, M. Shaposhnikov, Higgs Boson mass and new physics. JHEP **10**, 140 (2012). arXiv:1205.2893 [275 (2012)]

71. G. Degrassi, S. Di Vita, J. Elias-Miro, J.R. Espinosa, G.F. Giudice, G. Isidori, A. Strumia, Higgs mass and vacuum stability in the Standard Model at NNLO. JHEP **08**, 098 (2012). https://doi.org/10.1007/JHEP08(2012)098

72. I. Masina, Higgs boson and top quark masses as tests of electroweak vacuum stability. Phys. Rev. D **87**, 053001 (2013). arXiv:1209.0393

73. V. Branchina, E. Messina, Stability, Higgs boson mass, and new physics. Phys. Rev. Lett. **111**, 1–5 (2013). arXiv:1307.5193

74. D. Buttazzo, G. Degrassi et al., Investigating the near-criticality of the Higgs boson. JHEP **12**, 089 (2013). arXiv:1307.3536

75. L. Di Luzio, L. Mihaila, On the gauge dependence of the Standard Model vacuum instability scale. JHEP **06**, 079 (2014). arXiv:1404.7450

76. N.K. Nielsen, Removing the gauge parameter dependence of the effective potential by a field redefinition. Phys. Rev. D **90**, 036008 (2014). arXiv:1406.0788

77. A. Andreassen, W. Frost, M.D. Schwartz, Consistent use of the standard model effective potential. Phys. Rev. Lett. **113**, 241801 (2014). arXiv:1408.0292

78. J.R. Espinosa, G.F. Giudice et al., The cosmological Higgstory of the vacuum instability. JHEP **09**, 174 (2015). arXiv:1505.04825

79. A.V. Bednyakov, B.A. Kniehl, A.F. Pikelner, O.L. Veretin, Stability of the electroweak vacuum: gauge independence and advanced precision. Phys. Rev. Lett. **115**, 201802 (2015). arXiv:1507.08833

80. M. Lindner, Implications of triviality for the standard model. Z. Phys. C **31**, 295 (1986)

81. B. Schrempp, M. Wimmer, Top quark and Higgs boson masses: interplay between infrared and ultraviolet physics. Prog. Part. Nucl. Phys. **37**, 112 (1996). arXiv:hep-ph/9606386

82. G. Altarelli, G. Isidori, Lower limit on the Higgs mass in the standard model: an update. Phys. Lett. B **337**, 141–144 (1994)

83. N. Cabibbo, L. Maiani, G. Parisi, R. Petronzio, Bounds on the fermions and Higgs boson masses in grand unified theories. Nucl. Phys. B **158**, 295–305 (1979)

84. P.Q. Hung, Vacuum instability and new constraints on fermion masses. Phys. Rev. Lett. **42**, 873 (1979)

85. G. Aad, T. Abajyan et al., Observation of a new particle in the search for the Standard Model Higgs boson with the ATLAS detector at the LHC. Phys. Lett. B **716**, 1–29 (2012). arXiv:1207.7214

86. S. Chatrchyan, V. Khachatryan et al., Observation of a new boson at a mass of 125 GeV with the CMS experiment at the LHC. Phys. Lett. B **716**, 30–61 (2012). arXiv:1207.7235

87. V. Branchina, E. Messina, Stability and UV completion of the Standard Model. EPL 117(6), 61002 (2017). https://doi.org/10.1209/0295-5075/117/61002

88. L. Di Luzio, G. Isidori, G. Ridolfi, Stability of the electroweak ground state in the Standard Model and its extensions. Phys. Lett. B **753**, 150–160 (2016). arXiv:1509.05028

89. J.A. Casas, J.R. Espinosa, M. Quiros, Improved Higgs mass stability bound in the standard model and implications for supersymmetry. Phys. Lett. B **342**, 171 (1995). arXiv:hep-ph/9409458

90. J.A. Casas, J.R. Espinosa, M. Quirós, Standard model stability bounds for new physics within LHC reach. Phys. Lett. B **382**, 374–382 (1996). arXiv:hep-ph/9603227

91. G. Isidori, G. Ridolfi, A. Strumia, On the metastability of the Standard Model vacuum. Nucl. Phys. B **609**, 387–409 (2001). arXiv:hep-ph/0104016v2

92. C.P. Burgess, V. Di Clemente, J. Ramón Espinosa, Effective operators and vacuum instability as heralds of new physics. JHEP **1**, 041 (2002). arXiv:hep-ph/0201160

93. G. Isidori, V.S. Rychkov, A. Strumia, N. Tetradis, Gravitational corrections to standard model vacuum decay. Phys. Rev. D **77**, 1–6 (2008). arXiv:0712.0242

94. N. Arkani-Hamed, S. Dubovsky, L. Senatore, G. Villadoro, (No) eternal inflation and precision Higgs physics. JHEP **0803**, 075 (2008). arXiv:0801.2399

95. F. Bezrukov, M. Shaposhnikov, Standard model Higgs boson mass from inflation: two loop analysis. JHEP **0907**, 089 (2009). arXiv:0904.1537

96. L.J. Hall, Y. Nomura, A finely-predicted Higgs boson mass from a finely-tuned weak scale. JHEP **1003**, 076 (2010). arXiv:0910.2235

97. J. Ellis, J.R. Espinosa, G.F. Giudice, A. Hoecker, A. Riotto, The probable fate of the Standard Model. Phys. Lett. B **679**, 369–375 (2009). arXiv:0906.0954

98. F. Loebbert, J. Plefka, Quantum Gravitational Contributions to the Standard Model Effective Potential and Vacuum Stability. Mod. Phys. Lett. A30(34), 1550189 (2015). https://doi.org/10.1142/S0217732315501898

99. O. Czerwińska, Z. Lalak, Ł. Nakonieczny, Stability of the effective potential of the gauge-less top-Higgs model in curved spacetime. JHEP **11**, 207 (2015). https://doi.org/10.1007/JHEP11(2015)207

100. M. Gonderinger, Y. Li, H. Patel, M.J. Ramsey-Musolf, Vacuum stability, perturbativity, and scalar singlet dark matter. JHEP **1**, 53 (2010). arXiv:0910.3167

101. A. Drozd, B. Grzadkowski, J. Wudka, Cosmology of multi-singlet-scalar extensions of the standard model. Acta Phys. Pol. B **42**, 2255–2262 (2011). arXiv:1310.2985

102. C.-S. Chen, Y. Tang, Vacuum stability, neutrinos, and dark matter. JHEP **4**, 19 (2012). arXiv:1202.5717

103. H. Han, S. Zheng, New constraints on Higgs-portal scalar dark matter. JHEP **12**, 44 (2015). arXiv:1509.01765

104. S. Kanemura, M. Kikuchi, K. Yagyu, Radiative corrections to the Higgs boson couplings in the model with an additional real singlet scalar field. Nucl. Phys. B **907**, 286–322 (2016). arXiv:1511.06211

105. S. Coleman, Fate of the false vacuum: semiclassical theory. Phys. Rev. D **15**, 2929–2936 (1977)

106. E. Kolb, M.S. Turner, *The Early Universe* (Addison-Wesley Publishing Company, Redwood City, 1990)

107. K. Lee, E.J. Weinberg, Tunneling without barries. Nucl. Phys. B **267**, 181 (1986)

108. G.C. Callan, S. Coleman, Fate of the false vacuum II, first quantum corrections. Phys. Rev. D **16**, 1762–1768 (1977)

109. G. Degrassi, SM vacuum stability (2014). Retrieved from http://benasque.org/2014imfp/talks_contr/296_Degrassi.pdf

110. LHC Higgs Cross Section Working Group: J.R. Andersen et al. In *Handbook of LHC Higgs Cross Sections: 3. Higgs Properties*, ed. By S. Heinemeyer, C. Mariotti, G. Passarino, R. Tanaka (2013). https://doi.org/10.5170/CERN-2013-004

111. A. Bredenstein, A. Denner, S. Dittmaier, M.M. Weber, Precise predictions for the Higgs-boson decay $H \rightarrow WW/ZZ \rightarrow 4$ leptons. Phys. Rev. D **74**, 013004 (2006). arXiv:hep-ph/0604011

112. A. Bredenstein, A. Denner, S. Dittmaier, M.M. Weber, Radiative corrections to the semileptonic and hadronic Higgs-boson decays $H \rightarrow W W / Z Z \rightarrow 4$ fermions. JHEP **02**, 080 (2007). arXiv:hep-ph/0611234

113. G. Belanger, B. Dumont, U. Ellwanger, J.F. Gunion, S. Kraml, Global fit to Higgs signal strengths and couplings and implications for extended Higgs sectors. Phys. Rev. D **88**, 075008 (2013). arXiv:1306.2941

114. D. Das, U. Ellwanger, A.M. Teixeira, NMSDECAY: a Fortran code for supersymmetric particle decays in the next-to-minimal supersymmetric standard model. Comput. Phys. Commun. **183**, 774–779 (2012). arXiv:1106.5633

115. LHC Higgs Cross Section Working Group: S. Dittmaier et al, Handbook of LHC higgs cross sections: 1. Inclusive Observables (2011). https://doi.org/10.5170/CERN-2011-002

116. The ATLAS, CDF, CMS, D0 Collaborations:, First combination of Tevatron and LHC measurements of the top-quark mass (2014). arXiv:1403.4427

117. ATLAS, CMS: G. Aad et al., Combined measurement of the Higgs boson mass in pp collisions at $\sqrt{s} = 7$ and 8 TeV with the ATLAS and CMS experiments. Phys. Rev. Lett. **114**, 191803 (2015). arXiv:1503.07589

118. Particle Data Group: K.A. Olive et al., Review of particle physics, update to Ref. [61] (2015). http://pdg.lbl.gov/2015/tables/rpp2015-sum-gauge-higgs-bosons.pdf

119. S. Heinemeyer, W. Hollik, D. Stöckinger, A.M. Weber, G. Weiglein, Precise prediction for M(W) in the MSSM. JHEP **08**, 052 (2006). arXiv:hep-ph/0604147

120. S. Heinemeyer, W. Hollik, A.M. Weber, G. Weiglein, Z pole observables in the MSSM. JHEP **04**, 039 (2008). arXiv:0710.2972

121. S. Heinemeyer, W. Hollik, G. Weiglein, L. Zeune, Implications of LHC search results on the W boson mass prediction in the MSSM. JHEP **12**, 084 (2013). arXiv:1311.1663

122. O. Stål, G. Weiglein, L. Zeune, Improved prediction for the mass of the W boson in the NMSSM. JHEP **09**, 158 (2015). arXiv:1506.07465

123. K. Matchev, TASI lectures on precision electroweak physics, in Particle physics and cosmology: the quest for physics beyond the

standard model(s). in *Proceedings, Theoretical Advanced Study Institute, TASI 2002, Boulder, USA, June 3–28, 2002* (2004), pp. 51–98. arXiv:hep-ph/0402031

124. M. Davier, A. Hoecker, B. Malaescu, Z. Zhang, Reevaluation of the hadronic contributions to the muon g-2 and to $\alpha(M_Z^2)$. Eur. Phys. J. C **71**, 1515 (2011). arXiv:1010.4180

125. Particle Data Group, Berkeley: K. Nakamura et al., Review of particle properties. J. Phys. G **37**, 075021 (2010)

126. G.W. Bennett, B. Bousquet et al., Final report of the E821 muon anomalous magnetic moment measurement at BNL. Phys. Rev. D **73**, 072003(2006). arXiv:hep-ex/0602035

127. S.M. Barr, A. Zee, Electric dipole moment of the electron and of the neutron. Phys. Rev. Lett. **65**, 21–24 (1990) [Erratum: Phys. Rev. Lett. **65**, 2920 (1990)]

128. D. Stöckinger, Topical review: the muon magnetic moment and supersymmetry. J. Phys. G **34**, R45–R91 (2007). arXiv:hep-ph/0609168

129. S. Heinemeyer, W. Hollik, G. Weiglein, Electroweak precision observables in the minimal supersymmetric standard model. Phys. Rep. **425**, 265–368 (2006). arXiv:hep-ph/0412214