THE EUROPEAN
PHYSICAL JOURNAL C

CrossMark

Special Article - Tools for Experiment and Theory

GAMBIT: the global and modular beyond-the-standard-model inference tool

The GAMBIT Collaboration: Peter Athron[1,2], Csaba Balazs[1,2], Torsten Bringmann[3], Andy Buckley[4], Marcin Chrzaszcz[5,6], Jan Conrad[7,8], Jonathan M. Cornell[9], Lars A. Dal[3], Hugh Dickinson[10], Joakim Edsjö[7,8], Ben Farmer[7,8,a], Tomás E. Gonzalo[3], Paul Jackson[2,11], Abram Krislock[3], Anders Kvellestad[12,b], Johan Lundberg[7,8], James McKay[13], Farvah Mahmoudi[14,15,e], Gregory D. Martinez[16], Antje Putze[17], Are Raklev[3], Joachim Ripken[18], Christopher Rogan[19], Aldo Saavedra[2,20], Christopher Savage[12], Pat Scott[13,c], Seon-Hee Seo[21], Nicola Serra[5], Christoph Weniger[22,d], Martin White[2,11], Sebastian Wild[23]

[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] SUPA, School of Physics and Astronomy, University of Glasgow, Glasgow G12 8QQ, UK
[5] Physik-Institut, Universität Zürich, Winterthurerstrasse 190, 8057 Zurich, Switzerland
[6] H. Niewodniczański Institute of Nuclear Physics, Polish Academy of Sciences, 31-342 Kraków, Poland
[7] Oskar Klein Centre for Cosmoparticle Physics, AlbaNova University Centre, 10691 Stockholm, Sweden
[8] Department of Physics, Stockholm University, 10691 Stockholm, Sweden
[9] Department of Physics, McGill University, 3600 rue University, Montreal, QC H3A 2T8, Canada
[10] Minnesota Institute for Astrophysics, University of Minnesota, Minneapolis, MN 55455, USA
[11] Department of Physics, University of Adelaide, Adelaide, SA 5005, Australia
[12] NORDITA, Roslagstullsbacken 23, 10691 Stockholm, Sweden
[13] Department of Physics, Imperial College London, Blackett Laboratory, Prince Consort Road, London SW7 2AZ, UK
[14] Univ Lyon, Univ Lyon 1, ENS de Lyon, CNRS, Centre de Recherche Astrophysique de Lyon UMR5574, 69230 Saint-Genis-Laval, France
[15] Theoretical Physics Department, CERN, 1211 Geneva 23, Switzerland
[16] Physics and Astronomy Department, University of California, Los Angeles, CA 90095, USA
[17] LAPTh, Université de Savoie, CNRS, 9 chemin de Bellevue, B.P.110, 74941 Annecy-le-Vieux, France
[18] Max Planck Institute for Solar System Research, Justus-von-Liebig-Weg 3, 37077 Göttingen, Germany
[19] Department of Physics, Harvard University, Cambridge, MA 02138, USA
[20] Faculty of Engineering and Information Technologies, Centre for Translational Data Science, School of Physics, The University of Sydney, Sydney, NSW 2006, Australia
[21] Department of Physics and Astronomy, Seoul National University, 1 Gwanak-ro, Gwanak-gu, Seoul 08826, Korea
[22] GRAPPA, Institute of Physics, University of Amsterdam, Science Park 904, 1098 XH Amsterdam, The Netherlands
[23] DESY, Notkestraße 85, 22607 Hamburg, Germany

Abstract We describe the open-source global fitting package GAMBIT: the Global And Modular Beyond-the-Standard-Model Inference Tool. GAMBIT combines extensive calculations of observables and likelihoods in particle and astroparticle physics with a hierarchical model database, advanced tools for automatically building analyses of essentially any model, a flexible and powerful system for interfacing to external codes, a suite of different statistical methods and parameter scanning algorithms, and a host of other utilities designed to make scans faster, safer and more easily-extendible than in the past. Here we give a detailed description of the framework, its design and motivation, and the current models and other specific components presently implemented in GAMBIT. Accompanying papers deal with individual modules and present first GAMBIT results. GAMBIT can be downloaded from gambit.hepforge.org.

a e-mail: benjamin.farmer@fysik.su.se
b e-mail: anders.kvellestad@nordita.org
c e-mail: p.scott@imperial.ac.uk
d e-mail: c.weniger@uva.nl
e Also Institut Universitaire de France, 103 boulevard Saint-Michel, 75005, Paris, France

Contents

Springer

1 Introduction

The search for physics Beyond the Standard Model (BSM) is a necessarily multidisciplinary effort, as evidence for new physics could appear in any observable in particle, astroparticle or nuclear physics. Strategies include producing new particles at high-energy colliders [1–3], hunting for their influences on rare processes and precision measurements [4–6], directly detecting dark matter (DM) in the laboratory [7–9], carefully examining cosmological observations for the influence of new physics [10–12], and detecting high-energy particles from DM annihilation or decay [13–15]. In this context, exclusions have so far been just as valuable as apparent detections; many purported signals of new physics have appeared [16–24], often only to be disproven by a lack of correlated signals in other experiments [14,25–29].

Properly and completely weighing the sum of data relevant to a theory, from many disparate experimental sources, and making rigorous statistical statements about which models are allowed and which are not, has become a challenging task for both theory and experiment. This is the problem addressed by global fits: simultaneous predictions of a raft of different observables from theory, coupled with a detailed combined statistical analysis of the various experimental searches sensitive to them. Several attempts to address this problem have been made in particle physics, from the characterisation of electroweak physics with ZFitter [30] and later GFitter [31] to CKM fits [32], neutrino global fits [33–35] and global analyses of supersymmetry (SUSY) [36–38], in particular with packages like SuperBayeS [39–56], Fittino [57–59], MasterCode [60–71] and others [72–88].

BSM global fits today remain overwhelmingly focused on SUSY, specifically lower-dimensional subspaces of the minimal supersymmetric standard model (MSSM) [89–93], or, more rarely, the next-to-minimal variant (NMSSM) [94–97]. There are only a handful of notable exceptions for non-SUSY

models [98–107] and none for SUSY beyond the NMSSM. These studies, and their underlying software frameworks, were each predicated on one specific theoretical framework, relying on the phenomenologist-as-developer to hardcode the relevant observables and theory definitions. This inflexibility and the correspondingly long development cycle required to recode things to work with a new theory, are two of the primary reasons that global fits have been applied to so few other models. The unfortunate result has been that proper statistical analyses have not been carried out for most of the theories of greatest current interest. This is in spite of the fact that the LHC and other experiments have yet to discover any direct evidence for SUSY, heavily constraining the minimal variant [89–93]. It is therefore essential that as many new ideas as possible are rigorously tested with global fits.

Even working within the limited theoretical context for which they were designed, existing global fits do not offer a public framework that can be easily extended to integrate new observables, datasets and likelihood functions into the fit as they become available. Neither do they provide any standardised or streamlined way to deal with the complex interfaces to external codes for calculating specific observables or experimental likelihoods. Of the major SUSY global fit codes, only one (the now-discontinued SuperBayeS [113]) has seen a public code release, in stark contrast to many of the public phenomenological tools that they employ. Public code releases improve the reproducibility, accessibility, development and, ultimately, critique, acceptance and adoptance of methods in the community.

Another difficulty is that carrying out detailed joint statistical analyses in many-dimensional BSM parameter spaces is technically hard. It requires full understanding of many different theory calculations and experiments, considerable coding experience, large amounts of computing time, and careful attention to statistical and numerical methods [90,114–117]. Outside of global fits, the response has been to focus instead on individual parameter combinations or a limited, not-necessarily-representative part of the parameter space, e.g. [2,3,118]. Making concrete statements across ranges of parameters requires adopting either the Bayesian or frequentist statistical framework. These each impose specific mathematical conditions on how one discretely samples the parameter space and then combines the samples to make statements about continuous parameter ranges. The choice of statistical framework therefore has a strong bearing upon the appropriateness and efficiency of the scanning algorithm one employs [114,115]; random sampling is rarely adequate. Most global fits have so far assumed either Bayesian or frequentist statistics, discarding the additional information available from the other. They have also employed only a single parameter sampling algorithm each, despite the availability and complementarity of a wide range of relevant numerical methods.

Table 1 A GAMBIT reading list

User/reader	Wants	Should read sections/references
Cheersquad	To get an overview of features	2, referring to D
Playtime	To run GAMBIT	↪ A, 11.2–11.4, 12
Runtime	To run GAMBIT without causing injury	↪ 5.4, 6, 8, 9.1, 9.2, 10.2, 10.4, 10.7, 10.8
Dev	To add observables, backends, etc.	↪ all of 11 → 3, 4.1, 4.4, all of 10
Model-Dev	To add new theories	↪ all of 5
Guru	BOSS the world	↪ 4.5, 4.6, 7, 9.3
Physicist	Details of physics implemented	[108–111]
Stats/CompSci	Details of scanning algorithms & framework	[112]

Here we introduce GAMBIT, the Global And Modular BSM Inference Tool. GAMBIT is a global fitting software framework designed to address the needs listed above: theory flexibility, straightforward extension to new observables and external interfaces, code availability, statistical secularism and computational speed. In this paper we describe the GAMBIT framework itself in detail. First results for SUSY and the scalar singlet DM model can be found in accompanying papers [119–121], as can detailed descriptions of the constituent physics and statistics modules [108–112]. The GAMBIT codebase is released under the standard 3-clause BSD license,[1] and can be obtained from gambit.hepforge.org.

This paper serves three purposes. It is:

1. An announcement of the public release of GAMBIT,
2. A generally-accessible presentation of the novel and innovative aspects of GAMBIT, along with the possibilities it provides for future particle phenomenology,
3. A reference manual for the framework and associated code.

Goals 2 and 3 imply slightly different things for the structure and content of this paper. Here we begin by specifically addressing Goal 2, in Sect. 2. This section provides an extended synopsis of the flexible and modular design concept of GAMBIT, describing its main features and advances compared to previous global fits. Sect. 2 provides something of a teaser for the more extended 'manual', which can be found in Sects. 3–13. These sections describe how the features of GAMBIT are actually implemented, used and extended. A quick start guide can be found in Appendix A, library dependencies and supported compiler lists in Appendix B, specific SM parameter definitions in Appendix C, and a glossary of GAMBIT-specific terms in Appendix D. When terms make their first or other crucial appearances in the text, we cross-link them to their entries in the glossary.

Within the 'manual' part of the paper, Sect. 3 describes in detail how a physics module in GAMBIT works, Sect. 4 details the system GAMBIT uses for interfacing with external codes, and Sect. 5 covers the internal model database and its influence on analyses and the rest of the code. Section 6 explains the user interface to GAMBIT and documents the available settings in the master initialisation file. Section 7 details the GAMBIT system for instigating scans by automatically activating different calculations, depending on the models scanned and the observables requested by the user. Section 8 explains how GAMBIT deals with statistical and parameter scanning issues; further details of the specific methods and optimisation options in the scanning module can be found in Ref. [112]. Section 9 describes the system for outputting results from GAMBIT. Section 10 covers other assorted utility subsystems. Section 11 discusses the build and automatic component registration system, including a crash course in adding new models, observables, likelihoods, scanners and other components to GAMBIT. Section 12 describes some minimal examples included in the base distribution, and provides information about releases and support.

A code like GAMBIT and a paper such as this have multiple levels of user and reader. The relevant sections of this paper for each are summarised in Table 1. Those more interested in understanding what GAMBIT offers than actually running or extending it need only this introduction, Sect. 2 and the glossary (Appendix D). Users interested in running scans without modifying any code should find more than enough to get started in Appendix A, Sects. 11.2–11.4 and 12. To get the most out of the code, such users should then move progressively on to Sects. 5.4, 6, 8, 9.1, 9.2, 10.2, 10.4, 10.7 and 10.8. Those interested in adding new observables, likelihoods or interfaces to external codes should also read Sects. 3, 4.1, 4.4, and the rest of Sects. 10 and 11. Users wanting to extend GAMBIT to deal with new models and theories should add the remainder of Sect. 5 to this tally. Power users

---

[1] http://opensource.org/licenses/BSD-3-Clause. Note that fjcore [122] and some outputs of FlexibleSUSY [123] (incorporating routines from SOFTSUSY [124]) 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.
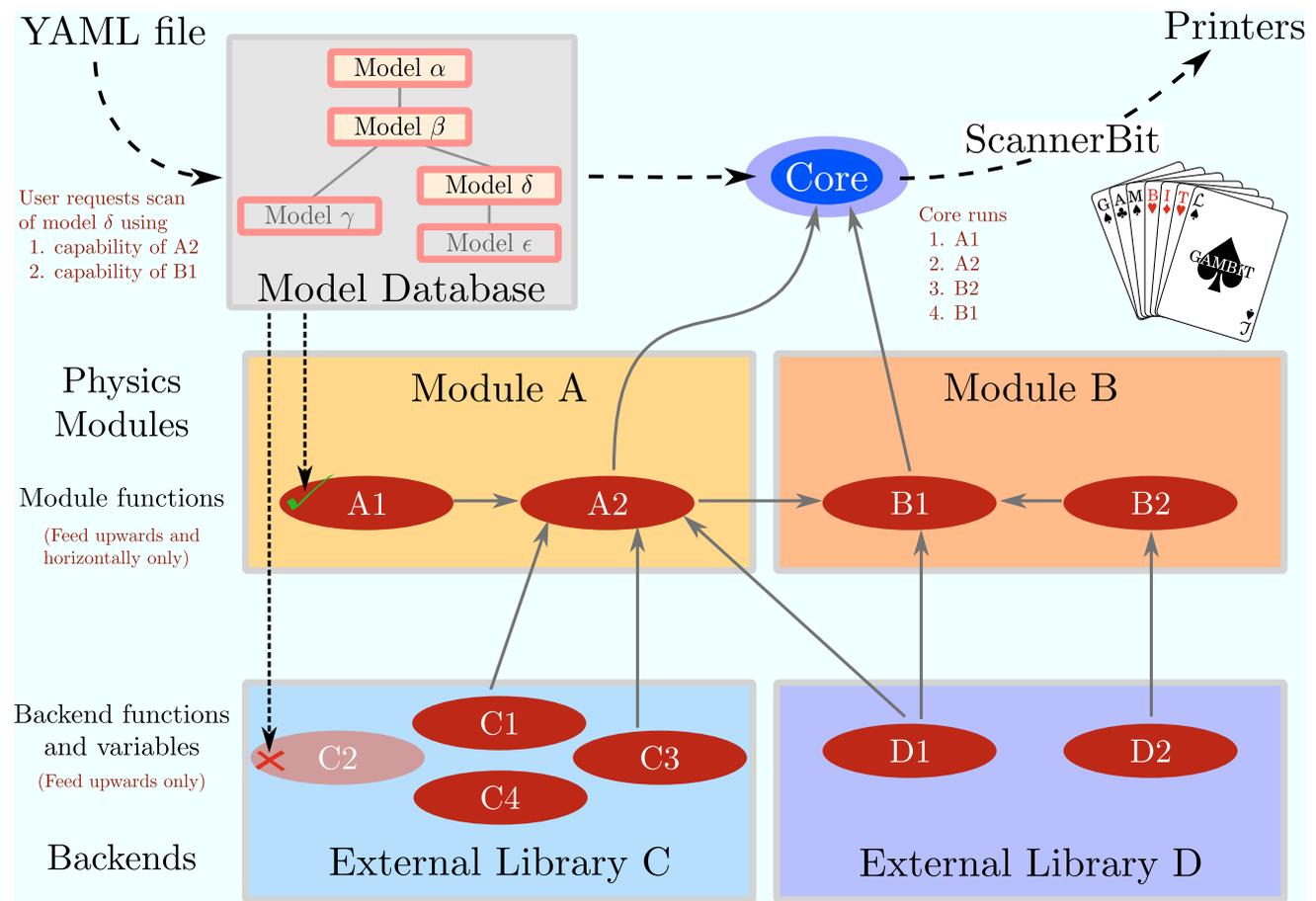
Fig. 1 A schematic representation of the basic elements of a GAMBIT scan. The user provides a YAML input file (see www.yaml.org), which chooses a model to scan and some observables or likelihoods to calculate. The requested model δ and its ancestor models (see text for definition) β and α are activated. All model-dependent module and backend functions/variables are tested for compatibility with the activated models; incompatible functions are disabled (C2 in the example). Module functions are identified that can provide the requested quantities (A2 and B1 in the example), and other module functions are identified to fulfil their dependencies. More are identified to fulfil those functions' dependencies until all dependencies are filled. Backend functions and variables are found that can fulfil the backend requirements of all chosen module functions. The Core determines the correct module function evaluation order. It passes the information on to ScannerBit, which chooses parameter combinations to sample, running the module functions in order for each parameter combination. The requested quantities are output by the printer system for each parameter combination tested

and developers wanting to have a complete understanding of the software framework should also familiarise themselves with Sects. 4.5, 4.6, 7 and 9.3. Readers and users with specific interests in particular physical observables, experiments or likelihoods should also add the relevant physics module paper(s) [108–111] to this list, and those interested in details of parameter scanning or statistics should likewise add Ref. [112].

2 Design overview

GAMBIT consists of a number of **modules** or 'Bits', along with various Core components and utilities. Figure 1 is a simplified representation of how these fit together. GAMBIT modules are each either **physics modules** (DarkBit, Collid-

erBit, etc.) or the scanning module, ScannerBit. ScannerBit is responsible for parameter sampling, prior transformations, interfaces to external scanning and optimisation packages and related issues; it is discussed in more detail in Sect. 8 and Ref. [112].

2.1 Modularity

2.1.1 Physics modules, observables and likelihoods

The first version of GAMBIT ships with six physics modules:

ColliderBit calculates particle collider observables and likelihoods. It includes detailed implementations of LEP, ATLAS and CMS searches for new particle production, and measurements of the Higgs boson. The LEP likeli-

hoods are based on direct cross-section limits on sparticle pair production from ALEPH, OPAL and L3. Fast Monte Carlo simulation of signals at ATLAS and CMS can be performed with a specially parallelised version of Pythia 8 [125]. ColliderBit offers the option to carry out detector simulation with BuckFast, a fast smearing tool, or the external package Delphes [126,127]. We have validated all likelihoods and limits via extensive comparison to experimental limits and cutflows. Higgs likelihoods in the first version of ColliderBit are provided exclusively by communication with HiggsBounds [128–130] and HiggsSignals [131]. Supersymmetric models are presently supported natively by the LEP and LHC likelihoods. The Higgs likelihoods are model-independent in as much as they require only Higgs couplings as inputs. Other models can be supported in LHC calculations by reading matrix elements into Pythia 8, e.g. from MadGraph [132,133]. For a detailed description, see [108].

FlavBit calculates observables and likelihoods from flavour physics, in particular B, D and K meson decays as observed by LHCb, including angular observables and correlations. Possibilities for inter-code communication exist with SuperIso [134–136] and FeynHiggs [137–143]. Supersymmetry is supported directly. A broad range of other models is supported, via the use of effective field theory. Likelihoods and observables have been validated by comparison to existing flavour fits [144–146]. See [109].

DarkBit calculates DM observables and likelihoods, from the relic abundance to direct and indirect searches. It includes an on-the-fly cascade decay spectral yield calculator, and a flexible, model-independent relic density calculator capable of mixing and matching aspects from existing backends, including DarkSUSY [147] and micrOMEGAs [148–153]. Direct detection likelihoods in DarkBit are based on calls to the DDCalc package [110]. Indirect detection can be carried out with the help of nulike [15] (neutrinos) and gamLike [110] (gamma rays). Validation of relic density calculations is based on extensive comparison with results from standalone versions of DarkSUSY and micrOMEGAs. Direct and indirect limits are validated by comparison with exclusion curves from the relevant experiments. All calculations support MSSM neutralinos and all other WIMPs (in particular, this includes Higgs portal models such as scalar singlet dark matter). See [110] for details.

SpecBit interfaces to one of a number of possible external spectrum generators in order to determine pole masses and running parameters, and provides them to the rest of GAMBIT in a standardised spectrum container format. Spectrum generators currently supported include FlexibleSUSY [123] and SPheno [154,155]. Models include MSSM models defined at arbitrary scales and the scalar singlet model. Support for additional spectrum generators and models is straightforward for users to add. Results of the existing code have been validated by comparison to standalone versions of FlexibleSUSY, SPheno and SOFTSUSY [124,156–159]. SpecBit also carries out vacuum stability calculations and perturbativity checks, which have been validated against existing results in the literature. See [111] for full details.

DecayBit calculates decay rates of all relevant particles in the BSM theory under investigation, and contains decay data for all SM particles. Theory calculations can make use of interfaces to FeynHiggs [137–143] and an improved version of SUSY-HIT [160–163], validated against direct SLHA communication with the same codes. DecayBit supports the MSSM and scalar singlet models. See [111].

PrecisionBit calculates model-dependent precision corrections to masses, couplings and other observables, as well as precision nuisance likelihoods for e.g. Standard Model (SM) parameters. BSM calculations are presently limited to the MSSM, and can call on GM2Calc [164], FeynHiggs [137–143] and SuperIso [134–136]. See [111].

Physics modules are collections of **module functions**, each capable of calculating a single physical or mathematical quantity. This may be an observable, likelihood component or any intermediate quantity required for computing one or more observables or likelihoods.

Each module function is tagged with a **capability**, which together with the associated **type** describes exactly what **quantity** it is capable of calculating. Module functions, rather than modules themselves, are the main building blocks of GAMBIT. The capability-type pairs associated with module functions are the threads that allow GAMBIT to automatically stitch together multiple functions into arbitrarily complicated physics calculations.

Individual module functions may have one or more **dependencies** on quantities that their own calculations depend on. At runtime, GAMBIT selects an appropriate module function to fulfil each dependency, by matching the declared capabilities of module functions with the declared dependencies of other module functions. This process also requires matching the declared return types of module functions with the types requested in each dependency.

A simple example is the W mass likelihood function in PrecisionBit, which has capability `lnL_W_mass`. This function calculates a basic χ^2 likelihood for the W mass, and is correspondingly named `lnL_W_mass_chi2`. To do its job, `PrecisionBit::lnL_W_mass_chi2` must be provided with a predicted value for the W mass, by some other module function in GAMBIT. These aspects are declared

```
#define CAPABILITY lnL_W_mass
START_CAPABILITY
  #define FUNCTION lnL_W_mass_chi2
  START_FUNCTION(double)
  DEPENDENCY(mw, triplet<double>)
  #undef FUNCTION
#undef CAPABILITY
```

Here the `DEPENDENCY` on the *W* mass `mw` is explicitly declared, and the declaration demands that it must be provided as a set of three real numbers, corresponding to a central value with upper and lower uncertainties (a `triplet<double>`). `lnL_W_mass_chi2` accesses these values in its actual source via a pointer named `mw` placed in a special `namespace` reserved for dependencies (`Pipes::lnL_W_mass_chi2::Dep`). It then uses the values to compute the likelihood, which it returns as its result:

```
/// W boson mass likelihood
const double mw_central_observed = 80.385;
const double mw_err_observed = 0.015;
void lnL_W_mass_chi2(double &result)
{
  using namespace Pipes::lnL_W_mass_chi2;
  double theory_uncert = std::max(Dep::mw->upper,
   Dep::mw->lower);
  result = Stats::gaussian_loglikelihood(
   Dep::mw->central, mw_central_observed,
   theory_uncert, mw_err_observed);
}
```

This module function has no concern for precisely *where* or *how* the *W* mass has been determined. This allows GAMBIT to choose for itself at runtime, on the basis of the model being scanned, whether it should provide e.g. an MSSM-corrected prediction (for an MSSM scan), or a different prediction (for a scan of a different model). This serves to illustrate the power of the modular design of GAMBIT, allowing different calculations to be automatically reused in myriad different physics scenarios, with essentially zero user intervention.

Section 3 covers declaring and writing module functions in detail.

2.1.2 Backends

External software packages that might be useful for calculating specific quantities are referred to in GAMBIT as **backends**. Examples of these might be DarkSUSY [147] (for, e.g., relic density calculations), or FeynHiggs [139–143] (for, e.g., Higgs mass calculations). A full list of existing codes with which GAMBIT can communicate via the backend system, along with all relevant references, can be found in the file `README.md` included in the main distribution. All studies that make use of GAMBIT with a backend must cite all the literature associated with that backend, along with all relevant GAMBIT literature.

Although GAMBIT itself is written in C++, with a small admixture of Python for build abstraction, backends can in principle be written in any language. Module functions can directly call **backend functions** and access global **backend variables** from these codes. To do this, a module function must declare that it has a **backend requirement**, which is then matched at runtime to the declared capability of a function or variable from some backend. This mirrors the manner in which GAMBIT fills dependencies from amongst the available module functions.

Whilst module functions can have both dependencies (resolvable with other module functions) and backend requirements (resolvable with backend functions or variables), backend functions and variables cannot themselves have either dependencies nor backend requirements. This is illustrated in the example in Fig. 1: backend functions and variables feed into module functions, but nothing feeds into the backend functions nor variables themselves.

A simple example is the calculation in DarkBit of the rate at which DM is gravitationally captured by the Sun:

```
#define CAPABILITY capture_rate_Sun
START_CAPABILITY
  #define FUNCTION capture_rate_Sun_const_xsec
  START_FUNCTION(double)
  DEPENDENCY(mwimp, double)
  DEPENDENCY(sigma_SI_p, double)
  DEPENDENCY(sigma_SD_p, double)
  BACKEND_REQ(cap_Sun_v0q0_isoscalar, (), double,
   (const double&, const double&, const double&))
  #undef FUNCTION
#undef CAPABILITY
```

Here `DarkBit::capture_rate_Sun_const_xsec` depends on the DM mass and scattering cross-sections, and explicitly declares that it requires access to a function from a backend. It demands that the backend function be tagged with capability `cap_Sun_v0q0_isoscalar`, that it take three `const double&` arguments, and that it must return a `double` result. The declaration of a matching backend function (taken in this example from the interface to DarkSUSY 5.1.3) would then look like:

```
BE_FUNCTION(dsntcapsuntab, double, (const double&,
 const double&, const double&), "dsntcapsuntab_",
 "cap_Sun_v0q0_isoscalar")
```

The function `DarkBit::capture_rate_Sun_const_xsec` then accesses the backend function from its source via a similar alias system to the one used for dependencies:
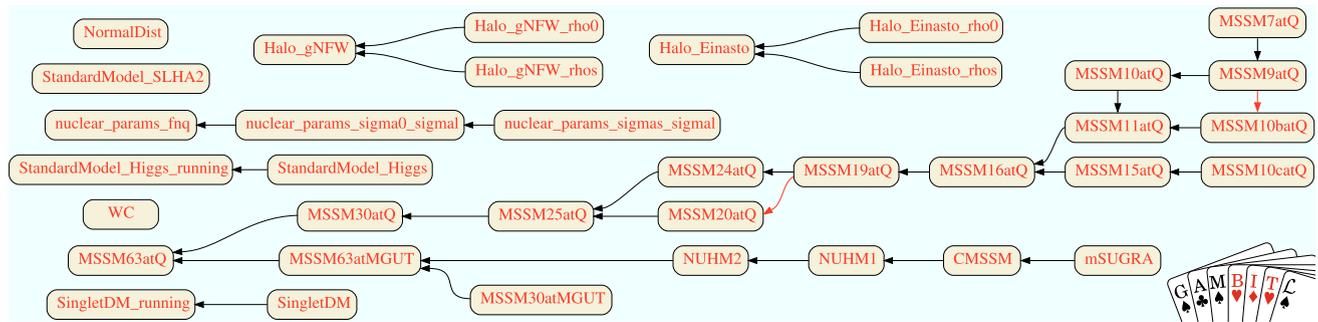
Fig. 2 The model hierarchy graph of the pre-defined models that ship with GAMBIT 1.0.0. The graph forms a set of disconnected directed trees, potentially linked by **friend** translation pathways. Nodes are individual **models**. Black arrows indicate **child**-to-**parent** translation pathways. The red arrows from MSSM9atQ to MSSM10batQ, and from MSSM19atQ to MSSM20atQ, indicate translations to **friend models**. Friend translations can cross between otherwise disconnected family trees, or, as in these two examples, between different branches of the same tree. Graphs like this (including any additional user-specified models) can be generated by running `gambit models` from the command line, and following the instructions provided

```
// Capture rate in s⁻¹ of regular DM in the Sun
// ( ⟹ σ is neither v-dependent nor
// q-dependent), assuming isoscalar interactions (σ_p = σ_n).
void capture_rate_Sun_const_xsec(double &result)
{
  using namespace
  Pipes::capture_rate_Sun_const_xsec;
  result =
  BEreq::cap_Sun_v0q0_isoscalar(*Dep::mwimp,
    *Dep::sigma_SI_p, *Dep::sigma_SD_p);
}
```

Typically, the requirement `cap_Sun_v0q0_isoscalar` will be fulfilled by DarkSUSY, a Fortran code – but there is nothing about this particular example function nor its declaration that forces such a pairing. The only conditions are that the selected backend function fulfils the requirements laid out in the `BACKEND_REQ` declaration. This is another example of the power of the modular design of GAMBIT, allowing it to attach any matching function from any backend at runtime, and to adapt to the presence or absence of different versions of different backends present on any given user's system.

There are many additional options and declarations available but not shown in this example, for constraining which versions of which backends are permitted to provide which backend requirement, under what model-dependent conditions and so on. Two additional features of note are not shown in Fig. 1: **backend initialisation functions**, which always run before any functions or variables in a backend are used, and **backend convenience functions**, which are agglomerations of functions and variables from one backend, presented to the rest of GAMBIT as if they are single backend functions.

Declaration of backend requirements is covered in detail in Sect. 3.1.3, and declaration of actual interfaces to backends is covered in Sect. 4.

2.1.3 Models

The models already implemented in GAMBIT 1.0.0 are shown in Fig. 2, and described in detail in Sect. 5.4. Instructions for adding new models are given in Sects. 5.1 and 11.1.

GAMBIT automatically activates or disables module and backend functions[2] according to their compatibility with the BSM model under investigation. It does this using a hierarchical model database, where each model is defined as a set of free parameters and a series of relations to other models. Models can be declared as children of existing models, which implies that there exists a mapping from the child parameter space to some subspace of the **parent** space. Each **child model** comes with a function that defines the transformation required to take a parameter point in its space to a corresponding point in the parent parameter space. GAMBIT uses these transformations at runtime to deliver the same parameter point in different parameterisations to different module functions, according to their declared needs. Models can also have translations pathways defined to other so-called **friend models** outside their own family tree.

One important aspect of this arrangement is that models can be arbitrarily 'bolted together' for any given scan, so that multiple models can be scanned over simultaneously, and their parameter values delivered together to any module functions that need them. This allows for the SM parameters to be varied as nuisance parameters when doing an MSSM or other BSM scan, for example. It also means that in such a joint SM-MSSM scan, the *same* underlying SM model (and therefore the same SM calculations wherever possible) will be used as in any other joint SM-BSM scan.

When a user requests a scan of a particular BSM model, that model and its entire model ancestry are activated. This

---

[2] and backend variables – but from here we will stop explicitly referring to backend functions and backend variables as different things except where it actually matters.

makes all module and backend functions that are compatible with any model in the activated ancestry available as valid building blocks of the scan. This provides maximum safety by forbidding any calculations that are not valid for the model under consideration, and maximum re-usability of modules, backends and their functions with new models, by providing certainty about which existing functions are 'safe' to use with new additions to the model hierarchy.

A basic example of model and backend function activation/deactivation can be seen in Fig. 1. Functions A1 and C2 have been specifically declared as model-dependent and therefore require activation or deactivation by the model database. Only functions that have been declared as model-dependent in this way are allowed to access the values of the underlying parameters in a scan. No other functions have any such declarations, so they are therefore valid for all models. Such functions *must* always work for any model, as all they need to do their job is to be confident that GAMBIT will deliver their declared dependencies and backend requirements in the form that they request – and GAMBIT guarantees precisely this for *all* module functions.

The two examples given in the previous subsections, of the *W* mass likelihood and the capture rate of DM by the Sun, are both examples of essentially model-independent calculations, where the module function does not need direct access to any of the underlying model parameters. These functions care only that their dependencies and backend requirements are available; if this is the case, they can do their jobs, irrespective of the underlying model actually being scanned.[3]

An example of an explicitly model-dependent module function is the DarkBit likelihood associated with the nuclear matrix elements relevant for spin-independent DM-nucleon scattering:

```
// Likelihoods for nuclear parameters.
START_CAPABILITY
  #define FUNCTION lnL_sigmas_sigma1
    START_FUNCTION(double)
    ALLOW_MODEL(nuclear_params_sigmas_signal)
  #undef FUNCTION
```

Here the `ALLOW_MODEL` declaration is used to indicate that the module function can *only* be used when scanning the

nuclear_params_sigmas_signal model (or one of its descendants – but it has none in this version of GAMBIT). This particular module function directly accesses the values of the model parameters, uses them to compute the joint likelihood and returns the result. In contrast, when the nuclear matrix elements are needed for calculating the physical DM-nucleon couplings in e.g., the scalar singlet Higgs portal model, they are instead upcast to the `nuclear_params_fnq` model (an ancestor of `nuclear_params_sigmas_signal`, cf. Fig. 2), and presented as such within the relevant module function:

```
#define FUNCTION DD_couplings_SingletDM
  START_FUNCTION(DM_nucleon_couplings)
  DEPENDENCY(SingletDM_spectrum, Spectrum)
  ALLOW_JOINT_MODEL(nuclear_params_fnq, SingletDM)
#undef FUNCTION
```

Here the `ALLOW_JOINT_MODEL` declaration explicitly forbids GAMBIT from using this function except when scanning *both* the `nuclear_params_fnq` and `SingletDM` models, or some pairwise combination of their respective descendants.

The GAMBIT model database, its declarations and features are discussed in much more detail in Sect. 5.

2.2 Adaptability and flexibility

After filtering out invalid module and backend functions by checking their compatibility with the model under investigation, GAMBIT works through the remaining functions to properly connect module functions to dependencies and backend functions to backend requirements. It starts with the quantities requested by the user (observables, likelihood components or other derived quantities), and then progressively resolves dependencies and backend requirements until it either reaches an impasse due to a mutual dependency between groups of module functions, or no outstanding needs remain. If all needs have been fulfilled, the result is a directed graph of dependencies, with no internal closed loops – a so-called directed acyclic graph. Directed acyclic graphs have the mathematical property that they possess an implied topological order. GAMBIT computes this ordering, and uses it to determine the optimal order in which to evaluate the module functions, such that each module function is guaranteed to run before any other function that depends on its results. GAMBIT further optimises the ordering beyond the constraint implied by this condition, considering the typical evaluation time of each function as a scan progresses, and its role in ruling out previous parameter combinations. We explain this overall **dependency resolution** process in detail in Sect. 7.

With a specific module function evaluation order in hand, GAMBIT passes the problem of actually sampling the param-

---

[3] Note the distinction between model-independent *functions* and model-independent *results*. Model-independent numerical results have the same values regardless of the physics model assumed. Model-independent functions act on input data according to the values of the data only, not according to the physics model that gave rise to the data. In general, the input data to model-independent functions are model-dependent quantities, leading to different results for different models. The *W* mass likelihood is a case in point: the predicted value of m_W and its likelihood are necessarily model-dependent quantities – but the function that computes the likelihood from a given value of m_W is not dependent on the model for which m_W has been computed.

eter space to ScannerBit (Sect. 8). ScannerBit engages whichever statistical scanning algorithm and output method a user has selected in their input file (see Sect. 6), choosing parameter combinations, calling the module functions in order, and sending the results to the GAMBIT printer system (Sect. 9). Functions log their activity via an extensive internal logging system (Sect. 10.2), and invalid parameter combinations, warnings and errors are identified using a dedicated exceptions system (Sect. 10.3).

This rather abstract formulation of the global fit problem enables a very high degree of automation, in turn providing flexibility and extendibility. By deferring the actual choice of the function that will provide the requisite physical inputs to each step of a calculation, GAMBIT makes it easy to confidently swap or add functions to existing scans. It also makes such scans efficient, as only the calculations needed for a given scan are actually activated, and each calculation is guaranteed to run only once for each parameter combination. Linking this to a hierarchical model database then provides the means for GAMBIT to automatically adapt existing likelihood and observable calculations to new models, to the largest extent theoretically possible. New components of course need to be added when different physics is to be considered for the first time, but the level of automation allows the user to immediately identify the precise gaps in the theoretical chain in need of new work, rather then wasting time by coding almost identical functions for every new model. This is facilitated by extensive and informative error messages presented when a scan is attempted but some link in the dependency chain cannot be fulfilled. These messages explain, for example, when a given dependency cannot be filled by any known function, if a requisite backend appears to be missing, if appropriate functions seem to exist but are not compatible with the model being scanned, if multiple permitted options exist for resolving a given dependency or backend requirement, and so on.

GAMBIT takes this flexibility and automatic adaptation even further by having the backend (Sect. 4) and build (Sect. 11) systems automatically add or disable modules, backends, models, printers and other components when new ones are defined, or when existing ones happen to be missing from a user's system. GAMBIT also includes extensive command-line diagnostics, which the user can employ to obtain reports on the status and contents of its components at many different levels (Sect. 10.4).

2.3 Performance and parallelisation

Parallelisation in GAMBIT happens at two levels: at the scanner level via MPI [165], and at the level of module functions with OpenMP [166]. This allows GAMBIT to easily

scale to many thousands of cores, as most major external sampling packages employ MPI, and a number of external physics codes make use of OpenMP (e.g. nulike [15] and forthcoming versions of DarkSUSY [147]). Users also have the option of implementing their own module functions using OpenMP natively in GAMBIT. In fact, GAMBIT can even automatically connect OpenMP-aware module functions and have other module functions run them in parallel using OpenMP. Section 3.1.4 explains how to achieve this. With this method, the total runtime for a single MSSM parameter combination, even including explicit LHC Monte Carlo simulation, can be reduced to a matter of a few seconds [108].

The performance of GAMBIT is explored in detail in the ScannerBit paper [112].

2.4 Available examples

In Sect. 12.1 we provide a series of examples showing how to run the full GAMBIT code. Any GAMBIT module can also be compiled with a basic driver into a standalone program. We also give a number of examples of module standalone drivers in Sect. 12.1, as well as dedicated examples for different modules included in first release [108–112].

A standalone driver program that calls a GAMBIT module needs to do a number of specific things:

– specify which model to work with,
– choose what the parameter values should be,
– indicate which module functions to run and what to do with the results,
– indicate which module functions to use to fulfil which dependencies,
– indicate which backend functions or variables to use to fulfil which backend requirements, and
– set input options that different module functions should run with.

These are all functions that are normally done automatically by GAMBIT. We provide a series of simple utility functions specifically designed for use in standalone driver programs though, allowing most of these operations to be completed in a single line each.

3 Modules

Other than the six physics and one scanning module in GAMBIT 1.0.0, behind the scenes GAMBIT also arranges backend initialisation functions into a virtual module known as BackendIniBit, and puts model parameter translation functions into effective modules of their own. These are discussed in detail in Sects. 4 and 5, respectively.

3.1 Module function declaration

GAMBIT modules and their functions are declared in a module's so-called **rollcall header**, using a series of convenient macros.

A module called *MyBit* would be created simply by creating a header `MyBit_rollcall.hpp` containing

```
#define MODULE MyBit
START_MODULE
#undef MODULE
```

and then rerunning the build configuration step in order to make GAMBIT locate the new file.[4]

Creating a module function requires a user to write it as a standard C++ function in a source file, and add a corresponding declaration to the rollcall header. The function should have return type void, and take exactly one argument by reference: the result of the calculation that the function is supposed to perform. This result can be of any type.[5] Taking a double-precision floating point number as an example, the definition of a function *function_name* in module *MyBit* would look like

```
namespace MyBit
{
  void function_name(double& result)
  {
    result = ... // something useful
  }
}
```

This would traditionally be placed in a file called `MyBit.cpp` or similar.

The declaration must state the name of the function, the type of its result, and the capability to assign to it. Such a declaration would look like

```
#define MODULE MyBit
START_MODULE
  #define CAPABILITY example_capability
  START_CAPABILITY
    #define FUNCTION function_name
    START_FUNCTION(double)
    #undef FUNCTION
  #undef CAPABILITY
#undef MODULE
```

where *example_capability* is the name of the capability assigned to the function *MyBit::function_name* in this example.

The following examples in Sects. 3.1.1–3.1.3 will show other specific declarations that may be given between `START_FUNCTION` and `#undef FUNCTION`.

---

[4] Re-running the configuration step is a generic requirement whenever adding new source or header files to GAMBIT. See Sect. 11 for details.

[5] At least, any type with a default constructor. Dealing in types without default constructors requires declaring objects internally in the module and returning pointers to them.

3.1.1 Model compatibility

In the absence of any specific declarations as to the model-dependency of the calculations in a module function, GAMBIT assumes that the function is completely model-independent. To instead declare that a module function may only be used with a single specific model *model_a*, one adds a declaration

```
ALLOW_MODEL(model_a)
```

after calling `START_FUNCTION`. To declare that the function may be used with one or more models from a particular set, one instead writes

```
ALLOW_MODEL(model_a)
ALLOW_MODEL(model_b)
...
```

or just

```
ALLOW_MODELS(model_a, model_b, ...)
```

where the ellipses ... indicate that the `ALLOW_MODELS` macro is variadic, and can take up to 10 specific models. Alternatively, to declare that *all* models from a given set must be in use, one declares

```
ALLOW_JOINT_MODEL(model_γ, model_δ, ...)
```

Declaring `ALLOW_MODEL`, `ALLOW_MODELS` or `ALLOW_JOINT_MODEL` also grants the module function access to the values of the parameters of the appropriate model(s) at runtime. Section 3.2.3 below deals with how to retrieve these parameter values.

GAMBIT is expressly designed for simultaneous scanning of multiple models, where the parameters of each model are varied independently. This allows for arbitrary combinations of different models, e.g. from including SM parameters as nuisances in a BSM scan, to varying cosmological and BSM parameters simultaneously in some early-Universe cosmological scenario. In these cases, module functions can be granted access to the parameters of multiple models at the same time, as long as the function is declared from the outset to need all of those parameters in order to operate correctly.

To set rules that constrain module functions' validities to scans of specific combinations of models, rather than simply declaring valid combinations one by one with `ALLOW_JOINT_MODEL`, a more involved syntax is required. Here, the possible individual models involved in the combinations are first listed with `ALLOW_MODEL_DEPENDENCE`. They are then placed into one or more specific **model groups**. Each allowed model combination is then specified by setting allowed combinations of model groups. If a given scan includes one model from each group listed in an allowed combination, then the module function is deemed to be compatible with the given scan.

For example, to specify that a function may be used when either *model_a* or *model_b* is being scanned, *but only if* *model_c* *is also being scanned at the same time*, one must write

```
ALLOW_MODEL_DEPENDENCE(model_a, model_b, model_c)
MODEL_GROUP(group1, (model_a, model_b))
MODEL_GROUP(group2, (model_c))
ALLOW_MODEL_COMBINATION(group1, group2)
```

This reveals that `ALLOW_JOINT_MODEL(`*model_γ*`,` *model_δ*`,` `...)` is simply a special case of this extended syntax, precisely equivalent to

```
ALLOW_MODEL_DEPENDENCE(model_γ, model_δ, ...)
MODEL_GROUP(group1, (model_γ))
MODEL_GROUP(group2, (model_δ))
...
ALLOW_MODEL_COMBINATION(group1, group2, ...)
```

Note that GAMBIT still deems a model to be in use even if its parameters are fixed to constant values during a scan. Declaring that a module function requires some model or model combination to be in use therefore merely demands that the model parameters have definite values during a scan, not that they are necessarily varied.

An explicit example of the syntax described in this section can be found in the declaration of the function `DarkBit::DD_couplings_MicrOmegas` in `DarkBit/include/gambit/DarkBit/DarkBit_rollcall.hpp`:

```
ALLOW_MODEL_DEPENDENCE(nuclear_params_fnq,
 MSSM63atQ, SingletDM)
MODEL_GROUP(group1, (nuclear_params_fnq))
MODEL_GROUP(group2, (MSSM63atQ, SingletDM))
ALLOW_MODEL_COMBINATION(group1, group2)
```

This function computes couplings relevant for direct detection, using microOMEGAs [153]. To do this, it needs the parameters of the nuclear matrix element model **nuclear_params_fnq**, plus the parameters of a dark matter model, which in GAMBIT 1.0.0 may be either the MSSM or the scalar singlet model.

3.1.2 Dependencies

To indicate that a module function requires some specific quantity as input in order to carry out its own calculation, one must declare that it has a dependency upon the capability, and the corresponding type, of some other module function. Dependencies are explicitly defined in terms of capabilities, not specific functions: *from the* GAMBIT *perspective functions do not depend on each other, they depend on each others' capabilities*. This is specifically designed to make module functions genuinely modular, by keeping the use of a module function's result completely independent of its identity. This has the (entirely intentional) consequence of making it practically impossible to safely use global states for passing information between module functions.

The syntax for declaring that a module function *function_name* has a dependency on some capability *capability* is simply to add a line

```
DEPENDENCY(capability, type)
```

to the module function declaration. Here *type* is the actual C++ type of the capability that needs to be available for *function_name* to use in its function body.

Such a declaration ensures that at runtime, GAMBIT will arrange its **dependency tree** such that it

(a) only runs *function_name* after some other module function with capability *capability* and return type *type* has already run for the same parameter combination,
(b) delivers the result of the other module function to *function_name*, so that the latter can use it in its own calculation.

It is also possible to arrange **conditional dependencies** that only apply when specific conditions are met. The simplest form is a purely model-dependent conditional dependency,

```
MODEL_CONDITIONAL_DEPENDENCY(capability, type,
 model_α, model_β, ...)
```

which would cause a function to depend on *capability* only when *model_α* and/or *model_β* is being scanned. Here the ellipses again indicate that up to 10 models can be specified.

A concrete example of this is the declaration of the function `FlavBit::SuperIso_modelinfo` in `FlavBit/include/gambit/FlavBit/FlavBit_rollcall.hpp`. This function is responsible for constructing the data object that will be sent to SuperIso [134,135] to tell it the values of the relevant Lagrangian parameters. Its declaration includes the lines:

```
MODEL_CONDITIONAL_DEPENDENCY(MSSM_spectrum,
 Spectrum, MSSM63atQ, MSSM63atMGUT)
MODEL_CONDITIONAL_DEPENDENCY(SM_spectrum,
 Spectrum, WC)
```

These statements cause the function to have a dependency on an `MSSM_spectrum` when scanning the MSSM, but a dependency on an `SM_spectrum` when scanning a low-energy effective theory of flavour (**WC**; see Sect. 5.4.5).

An alternative formulation allows both model conditions and backend conditions to be specified:

```
#define CONDITIONAL_DEPENDENCY capability
START_CONDITIONAL_DEPENDENCY(type)
ACTIVATE_FOR_MODELS(model_α, model_β, ...)
ACTIVATE_FOR_BACKEND(requirement, be_name1)
ACTIVATE_FOR_BACKEND(requirement, be_name2)
#undef CONDITIONAL_DEPENDENCY
```

In this example, the dependency on *capability* would not only be activated if *model_α* or *model_β* were in use, but also if

either backend *be_name1* or backend *be_name2* were used to resolve the backend requirement *requirement*. In this case, the `CONDITIONAL_DEPENDENCY` declaration must appear *after* the corresponding backend requirement is declared. Declaration of backend requirements is covered in Sect. 3.1.3.

There is currently no way to specify more complicated arrangements like 'dependency is activated only if scanning *model_α* and using *backend_name*' or 'only if scanning both *model_α* and *model_β*'. Wanting to use such complicated scenarios is usually a sign that the intended design of the module function is unnecessarily complicated, and the function would be better just split into multiple functions with different properties.

3.1.3 Backend requirements

Backend requirements are declarations that a module function intends to use either a function or a global variable from a backend (external) code. Backend requirements are specified in a similar way to dependencies: by declaring the type and the capability of the required backend function or variable (*not* the name of a specific backend function). In contrast to dependencies, however, the type of a backend requirement may be an entire function signature, describing not just the return type, but also the types of an arbitrary number of arguments. Designating the capability of the backend variable required as *var_requirement* and its required type *var_type*, the declaration of a backend variable requirement is

```
BACKEND_REQ(var_requirement, (tags), var_type)
```

If a backend function is required, with capability *fn_requirement*, return type *fn_return_type* and function argument types *arg1_type, arg2_type* and so on, the declaration is instead

```
BACKEND_REQ(fn_requirement, (tags), fn_return_type,
  (arg1_type, arg2_type, ...))
```

Note that the final argument of `BACKEND_REQ` should be absent for backend variable requirements, but should be explicitly specified as `()` for backend functions with no arguments – as is standard C/C++ syntax. The ellipses in the backend function example again indicate that the entry is variadic, so as many function arguments can be specified as required. If the backend function is *itself* required to be variadic (in the C-style sense that the function required must be able to take a variable number of arguments), then instead of the traditional ellipses used to declare such a function, one must use the keyword `etc`, as in

```
BACKEND_REQ(fn_requirement, (tags), fn_return_type,
  (arg1_type, etc))
```

The *tags* entry in the declarations above allows one to specify a set of zero or more comma-separated tags, which can then be used to impose various conditions on how backend requirements can be filled. Consider the following example:

```
BACKEND_REQ(req_A, (tag1), float, (int, int))
BACKEND_REQ(req_B, (tag1, tag2), int, ())
BACKEND_REQ(req_C, (tag3), int)
ACTIVATE_BACKEND_REQ_FOR_MODELS( (model_α,
  model_β), (tag1) )
BACKEND_OPTION( (be_name1), (tag1) )
BACKEND_OPTION( (be_name2, 1.2, 1.3, 1.5),
  (tag2, tag3) )
FORCE_SAME_BACKEND(tag1)
```

In this example, the `ACTIVATE_BACKEND_REQ_FOR_MODELS` directive ensures that *req_A* and *req_B* only exist as backend requirements when *model_α* and/or *model_β* are in use. `FORCE_SAME_BACKEND` creates a rule that at runtime, both *req_A* and *req_B* must be filled using functions from the same version of the same backend.

Further rules are given by the `BACKEND_OPTION` declarations. The first of these indicates that *be_name1* is a valid backend from which to fill one or both of *req_A* and *req_B*. The second `BACKEND_OPTION` declaration indicates that *req_B* and *req_C* may each be filled from versions 1.2, 1.3 or 1.5 only of *be_name2*. Version numbers here are both optional and variadic. Failure to list any version is taken to imply that any version of the backend is permitted. Presently there is no mechanism for indicating that only specific ranges of version numbers are permitted, short of listing each one explicitly. Version numbers can actually be specified in the same way when `ACTIVATE_FOR_BACKEND` is specified within a `CONDITIONAL_DEPENDENCY` declaration.

When *model_α* or *model_β* is being scanned, the rules in this particular snippet have the effect of forcing *req_A* to be filled from some version of *be_name1* (due to the first `BACKEND_OPTION` declaration), which in turn forces *req_B* to be filled from the same version of *be_name1* (due to the `FORCE_SAME_BACKEND` directive). If other models are scanned, *req_A* and *req_B* are simply ignored, and go unfilled. *Req_C* is forced to be filled from either version 1.2, 1.3 or 1.5 of *be_name2*, regardless of which models are scanned.

As with other GAMBIT rollcall header commands, the lists of models and tags in all backend requirement declarations are variadic. In this case there is practically no limit to the number of entries that a tag or model list may contain. Empty lists `()` are also permitted.

When a backend requirement has a rule imposed on it by one or more `BACKEND_OPTION` declarations, one of the stated options *must* be used. When none of the tags of a given backend requirement is mentioned in a `BACKEND_OPTION` command, any version of any backend is permitted as long as the capability and type match. Simply omitting `BACKEND_OPTION` altogether means that any matching function can be used, from any backend.

3.1.4 Parallel module functions

GAMBIT can make effective use of OpenMP parallelistaion either at the backend level, or natively within its own module functions. The simplest way to use OpenMP at the module function level is to place OpenMP directives inside a single module function, keeping the OpenMP block(s) wholly contained within the module function. In this case no special declarations are needed at the level of the module's rollcall header.

An alternative method is to have a single module function open and close an OpenMP block, and to call other module functions (indirectly) within that block, potentially very many times over for a single parameter combination. In this case we refer to the managing module function as a **loop manager** and the functions it calls **nested module functions**. Loop managers are declared using the `CAN_MANAGE_LOOPS` switch

```
START_FUNCTION(type, CAN_MANAGE_LOOPS)
```

Unlike regular module functions, loop managers may have *type* = `void`. Nested functions need to declare the capability of the loop manager that they require with

```
NEEDS_MANAGER_WITH_CAPABILITY(management_cap)
```

This declaration endows the function with a special dependency on *management_cap* that can only be fulfilled by a function that has been declared as a loop manager. The result type of the loop manager is ignored, i.e. loop managers of any return type are equally valid sources of this dependency.

This arrangement allows GAMBIT's **dependency resolver** to dynamically string together various nested module functions and instruct loop managers to run them in parallel. At runtime, nested functions are arranged into their own mini dependency trees, and pointers to ordered lists of them are handed out to the designated loop managers.

Other functions can depend on nested functions in the regular way. In this case they receive the final result of the function, the last time it is called by its loop manager for a given parameter combination. Loop managers are assigned hidden dependencies at runtime by the dependency resolver, on all quantities on which their designated nested functions depend. This ensures that a loop is not invoked until the dependencies of all functions in the loop have been satisfied.

The GAMBIT Core does not invoke any nested functions itself; this is the express responsibility of loop managers. The only exception to this rule occurs when for whatever reason a nested function's loop manager executes zero iterations of the loop it manages, but some other module function outside the loop depends on one of the nested functions that never ran; in this case the nested function is run the first time the dependent function tries to retrieve its value (as are any other nested functions that the first nested function depends on).

3.1.5 One-line module function declaration

It is also possible to declare a module function with its allowed models and even dependencies, in a single line:

```
QUICK_FUNCTION( module_name,
    example_capability,
    capability_vintage,
    function_name,
    function_type,
    (model_α, model_β, ...),
    (dep_cap1, dep_type1),
    (dep_cap2, dep_type2),
    ... )
```

Here one gives the module name explicitly, meaning that the declaration can even be used after `MODULE` has been `#undef`-ed. The argument *capability_vintage* tells GAMBIT whether or not *example_capability* has been declared previously; this can be set to either `NEW_CAPABILITY` or `OLD_CAPABILITY`. As usual, the variadic allowed model list (*model_α*, *model_β*, ...) can take up to 10 entries. This can be followed by up to 10 dependencies, given as capability-type pairs. The model list and dependency entries are optional arguments; specifying dependencies but leaving the allowed models free requires giving () for the allowed model list.

3.2 Pipes

Module functions must be entirely self-contained for GAMBIT to safely place them in a dependency tree. They must not call each other directly, nor call functions from specific backends directly. They should also strongly avoid setting or reading any global variables, especially those where the order of read or write operations might matter at all. The only safe way for code inside module functions to communicate with the outside world is via the function's own personal set of **pipes**.

At runtime, GAMBIT's **dependency resolver** (Sect. 7) connects each pipe to the relevant data stream that a module function is permitted to interact with. This might be the result of a module function deemed appropriate for fulfilling a dependency, a backend function fitting a backend requirement, or some other more specific utility.

Pipes are safe pointers, automatically declared when module functions themselves are declared. They and the data they point to can be set by the dependency resolver, but not by code inside module functions (except for the special case of data pointed to by a backend variable requirement pipe). They reside in the namespace `Pipes::`*function_name*.

Here we give a complete list of all the pipes that can be available to a module function, along with information on their usage and the circumstances under which they should be expected to exist.

3.2.1 Accessing dependencies

A dependency on a *capability* of *dep_type* can be accessed at runtime through the safe pointer

```
Pipes::function_name::Dep::capability
```

by simply dereferencing it, or calling *some_member_function* of class *dep_type*

```
using namespace Pipes::function_name;
dep_type my_variable = *Dep::capability;
Dep::capability->some_member_function();
```

e.g. if the function `decay_width` had a double-precision dependency on `mass`, one would simply type

```
double m = *Pipes::decay_width::Dep::mass;
```

The actual host module, name, capability and type of the function providing a dependency can be ascertained from its pipe, e.g.

```
using namespace Pipes::decay_width;
std::string m_module    = Dep::mass->origin();
std::string m_function  = Dep::mass->name();
std::string m_capability= Dep::mass->capability();
std::string m_type      = Dep::mass->type();
```

3.2.2 Accessing backend requirements

Backend requirements can be used or retrieved by way of the safe pointer

```
Pipes::function_name::BEreq::requirement
```

Take the example of a double-precision backend variable with capability `my_var_req`, declared in function `my_func` with

```
BACKEND_REQUIREMENT(my_var_req, (), double)
```

This variable is accessed directly as

```
using namespace Pipes::my_func;
double y = 2.5 + *BEreq::my_var_req;
*BEreq::my_var_req = y*y;
```

In the case of a backend function, e.g. declared as

```
BACKEND_REQUIREMENT(my_fn_req1, (), double,
 (double))
```

one can call the corresponding backend function by writing

```
using namespace Pipes::my_func;
double f_of_pi = BEreq::my_fn_req1(3.14159);
```

If necessary, the actual underlying function or variable pointer can be retrieved from a backend requirement pipe, by calling its `pointer()` method. This can be useful if a module or backend function requires a pointer to some function in order to perform its duties, as in the following example from `DarkBit::nuyield_from_DS`

```
// Hand back the pointer to the DarkSUSY
// neutrino yield function
result.pointer = BEreq::nuyield.pointer();
```

There is an important final subtlety to note here: because the arguments are forwarded through a number of different layers of indirection, in order to support the direct use of literals in calls to backend functions it is necessary to indicate explicitly if any non-literal parameters must be passed by value. The way to do this is to wrap such arguments in the helper function `byVal()`. For example, take a backend requirement of a function `my_func` declared as

```
BACKEND_REQUIREMENT(my_fn_req2, (), double,
 (double, double&))
```

This can be safely called as

```
using namespace Pipes::my_func;
double x = 2.0;
double y = BEreq::my_fn_req2(3.0, x);
```

or

```
using namespace Pipes::my_func;
double x = 2.0;
double y = BEreq::my_fn_req2(byVal(x), x);
```

but will fail to compile if

```
using namespace Pipes::my_func;
double x = 2.0;
double y = BEreq::my_fn_req2(x, x);
```

is attempted. The backend requirement system in GAMBIT is entirely typesafe, so if the code compiles one can at least be confident that the types in calls to backend functions correctly match their declarations.

As with dependencies, the name, capability and type of the backend function fulfilling a backend requirement can be extracted from its pipe, along with the host backend and its version, e.g.

```
using namespace Pipes::my_func;
std::string r3_function    = BEreq::r3->name();
std::string r3_capability  =
 BEreq::r3->capability();
std::string r3_type        = BEreq::r3->type();
std::string r3_backend     = BEreq::r3->origin();
std::string r3_bkend_versn = BEreq::r3->version();
```

3.2.3 Accessing model parameters

Model parameters are only provided to module functions that have been explicitly declared as model-dependent, and then only for the models actually being used in a particular scan. A module function is model dependent if it features an

ALLOWED_MODELS or ALLOW_MODEL_DEPENDENCE declaration, a model-conditional dependency, or a backend requirement activation rule that is conditional on some model. Once again, this is to enforce modularity; functions that claim to be model-independent through their (lack of) model declarations must operate only on dependencies and backend requirements, i.e. without using the values of the scanned parameters.

For module functions that are permitted access to the parameter values, all parameters of all models are delivered in a simple map of parameter names to their values. For such a function *function_name*, the value of a parameter *parameter_name* can then be retrieved with

```
double p = Pipes::function_name::Param["parameter_
  name"];
```

Whether or not the Param map contains a given parameter depends on whether or not its model is actually being scanned. This can be checked with the funtion

```
bool Pipes::function_name::ModelInUse(str);
```

which takes as input a string containing the model in question (str is just a typedef of std::string). Note that the models in use in different functions may not always be what one expects – the nature of the GAMBIT model hierarchy is such that if a module function declares that it can work with a model that is an *ancestor* of the actual model being scanned, the function will be permitted to run but will receive each parameter point delivered *in terms of the parameters of the ancestor model*, rather than directly in the parameters of the model actually being scanned.[6] This is an important feature, as it allows module functions to be re-used unaltered with models that may not have even been invented when the original module function was written.

Although it is possible to scan two models containing a parameter with a common name, it is not possible to retrieve both parameters from the Param map in the same module function. By default, GAMBIT raises a runtime error if models with common parameters are declared as allowed (by ALLOWED_MODELS or ALLOW_MODEL_DEPENDENCE) in a single module function, and then activated together in a scan. More adventurous users may wish to deactivate this error and allow such parameter clashes in some very specific circumstances (see Sect. 6.3.1).

---

[6] Note that if a module function is explicitly declared to work with *multiple* ancestors of the model being scanned, then only the parameters of the least-distant ancestor will be delivered. These rules also apply for activation of model-dependent depedencies and backend requirements (cf. Sects. 3.1.2, 3.1.3).

3.2.4 Accessing options from the input file

GAMBIT features an extensive system for specifying run options for module functions, discussed in detail in Sect. 6. Module functions access these options via a dedicated pipe, which connects to a miniature YAML structure generated by the dependency resolver from all the entries in the original input YAML file that actually apply to the module function in question.

The pipe is runOptions. It can be queried for the presence of a given option "my_option"

```
using namespace Pipes::function_name;
if (runOptions->hasKey("my_option"))
{
  // Do something exciting
}
```

or used to retrieve the value as a variable of type *opt_type*, either directly

```
using namespace Pipes::function_name;
opt_type x = runOptions->getValue<opt_type>
  ("my_option");
```

or with a default value *default*

```
using namespace Pipes::function_name;
opt_type x = runOptions->getValueOrDef<opt_type>
  (default, "my_option");
```

3.2.5 Managing parallel module functions

Running OpenMP loops containing GAMBIT module functions takes a little care, but it is ultimately one of the most efficient ways to speed up computationally challenging likelihood calculations.

A loop manager *lpman* is responsible for opening and closing the multi-threaded OpenMP block. Inside the block, it needs to use the void function

```
Pipes::lpman::Loop::executeIteration(long long)
```

to execute a single call to the chain of nested functions that it manages. Here the integer argument of the function is the iteration number, which is passed directly on to each nested function running inside the loop. A nested function *nested_fn* can access the iteration using the pipe iteration as

```
long long it = *Pipes::
  nested_fn::Loop::iteration;
```

Internally, GAMBIT holds the results of each module function in memory, for efficiently handing over results as dependencies and so on. For nested functions, it holds the results in an array of size equal to the number of threads. Serial module functions access the first element of this array

when retrieving dependencies, whereas nested module functions run by the same loop manager access the element corresponding to the thread number. This is what allows the nested module functions to run safely in parallel, in arbitrary dependency trees arranged by the dependency resolver at runtime.

A consequence of this setup is that any serial module function that depends on a nested module function will only read the result obtained in the last iteration of the first thread (i.e. of index 0). For this reason, it is generally advisable to run the final iteration of a GAMBIT OpenMP loop in serial, so as to properly sync the results for use further 'downstream'. Likewise, it is desirable to run the first iteration in serial as well, to allow any nested module functions to initialise any local static variables and other data elements that they might share across threads. With this consideration in mind, a minimal example of an OpenMP loop implemented in a loop manager is

```
using namespace Pipes::lpman;
Loop::executeIteration(0);
#pragma omp for
for (int i = 1; i < 9; i++)
{
  Loop::executeIteration(i);
}
Loop::executeIteration(9);
```

In this example, the first iteration of ten is run serially, the next 8 are done in parallel using however many threads are available, and the tenth and final iteration is again done serially.

The above example assumes that the number of required iterations is known at compile time. If this is not the case, one may call the void function pipe `wrapup()` from within a nested function, in order to signal to the loop manager that the loop can be terminated. When one of the nested module functions in one of the threads calls `wrapup()`, the boolean pipe

```
Pipes::lpman::Loop::done
```

in the function managing the loop is set to true, allowing it to cut the loop short. This allows constructions like

```
using namespace Pipes::lpman;
long long it = 0;
Loop::executeIteration(it);
#pragma omp parallel
{
  #pragma omp atomic
  it++;
  while(not *Loop::done)
  {
    Loop::executeIteration(it);
  }
}
Loop::executeIteration(it++);
```

to be used in loop managers. Note that using this pattern requires that it be safe for a few more iterations of the loop

to be performed after the `done` flag has been raised, because calling `wrapup()` in one thread will not affect other threads until they at least complete their own iterations and return to re-evaluate the while condition. The final serial iteration should generally also still be run as well, after the loop has terminated.

The `done` flag is automatically reset to `false` in all nested functions for each new parameter point. If for whatever reason it needs to be reset manually during a calculation, this can be achieved with the void function pipe

```
Pipes::lpman::Loop::reset()
```

which is available in all loop managers.

4 Backends

GAMBIT interfaces with backends by using them as runtime plug-ins. Backends are compiled into shared libraries, which GAMBIT then dynamically loads with the POSIX-standard `dl` library. This approach allows for direct access to the functions of the backend library and efficient data communication via memory, while at the same time keeping the build process of GAMBIT separate from of that of the particular backends used.

The locations of backend shared libraries can be specified in a YAML file `config/backend_locations.yaml`, with entries of the form

```
backend_name:
  backend_version: path_to_shared_library
```

Library paths can either be given as absolute paths, or relative to the main GAMBIT directory. If `backend_locations.yaml` does not exist, or if it is missing an entry for a given backend, GAMBIT will instead look for a path in the file `config/backend_locations.yaml.default`, which contains default library paths for all backends that GAMBIT has interfaces to.

Similar to module functions, functions in backend libraries are tagged with a **capability** describing what they can calculate. The capability tags are used by GAMBIT to match backend functions to the backend requirements declared by module functions. The layer of abstraction introduced by these tags allows appropriately designed module functions to use different backends interchangeably, given that they calculate the same quantity.

GAMBIT can currently communicate with backends written in C, C++ and Fortran. However, we must pay some attention to the differences between these languages. In particular, using a Fortran backend requires translating between standard Fortran and C-family types, and using a C++ backend typically involves loading entirely new types from the C++ library. We return to these topics in Sects. 4.4 and 4.5.

The interface to a backend library is declared in a frontend header file, located in

```
Backends/include/gambit/Backends/frontends
```

and named after the backend. Thus, a backend called MyBackend would be traditionally interfaced with GAMBIT via a frontend header `MyBackend.hpp`. To differentiate multiple versions of the same backend, the version number can be appended to the header name, e.g. `MyBackend_1_2.hpp` for version 1.2 of MyBackend. Applications such as this, where the periods in the version number are replaced with underscores, make use of what we refer to as the **safe version** of a backend, i.e. a representation of the version number that is safe to use when periods would be syntactically hazardous. A frontend header starts by defining the name, version and safe version of the backend, then immediately calls the `LOAD_LIBRARY` macro, which takes care of loading the backend shared library:

```
#define BACKENDNAME MyBackend
#define VERSION 1.2
#define SAFE_VERSION 1_2
LOAD_LIBRARY
```

4.1 Backend function declaration

The main pieces of information required to interface a backend function to GAMBIT are its return type, call signature and library symbol. The name mangling schemes of g++/gfortran and icpc/ifort (the two main compiler suites that GAMBIT supports; cf. Appendix B) are mostly consistent, so a single symbol name can usually be entered here for both compilers.[7] In addition, the function must be assigned a name and a capability. This is all specified via the `BE_FUNCTION` macro. For instance, a C/C++ backend function with the declaration

```
double getMatrixElement(int, int);
```

could be registered in the frontend header as

```
BE_FUNCTION(getMatrixElement, double, (int, int),
 "_Z13getMatrixElementii",
 "rotation_matrix_element")
```

---

[7] The symbols of a shared library, with names prepended by an additional underscore, can be obtained using the `nm` command. Functions within Fortran modules are an exception to the consistency of name mangling. The best way to deal with these is often to use the C-interoperability features of Fortran to explicitly choose a symbol name, taking the choice out of the hands of the compiler. An example of this can be seen in DDCalc [110]. In future, GAMBIT will automatically determine the appropriate name mangling itself, according to the scheme of the selected compiler.

where `"_Z13getMatrixElementii"` is the library symbol and `"rotation_matrix_element"` is the capability we assign to this function.

The macro `BE_VARIABLE` used to interface global variables in a backend library follows a similar syntax. If the backend contains a global variable

```
double epsilon;
```

controlling the tolerance of some calculation, it can be registered as

```
BE_VARIABLE(epsilon, double, "_epsilon",
 "tolerance")
```

with `"_epsilon"` the library symbol and `"tolerance"` the capability assigned to the variable.

Backend functions and variables can be declared as either model-independent or valid for use only with certain models, just like module functions can. The default is to treat everything as model-independent. To declare an alternative default that applies to an entire backend, one places

```
BE_ALLOW_MODELS(model_α, model_β, ...)
```

directly after `LOAD_LIBRARY`. This has the effect of allowing the entire backend to be used only if one or more of the listed models is involved in a scan. This default can be further overridden at the level of individual backend variables and backend functions, by adding additional model arguments to their declarations:

```
BE_FUNCTION(getMatrixElement, double, (int,int),
            "_Z13getMatrixElementii",
            "rotation_matrix_element",
            (model_α, model_β, ...) )
BE_VARIABLE(epsilon, double,
            "_epsilon",
            "tolerance",
            (model_α, model_β, ...) )
```

4.2 Convenience functions

If several backend function calls or variable manipulations are commonly performed together, it may be useful to combine these into a single backend convenience function. Although technically defined inside GAMBIT, this function will appear to the rest of GAMBIT as if it were simply another function in the backend library. Convenience functions are registered in the frontend header with the `BE_CONV_FUNCTION` macro. The syntax is identical to that of `BE_FUNCTION` except that there is no need to specify a library symbol, as the convenience function is not actually part of the backend library.

```
BE_CONV_FUNCTION(getMatrix, Matrix<double,2,2>,
 (), "full_rotation_matrix")
```

The definition of the convenience function can then either be given directly in the frontend header, or in a separate source file named after the backend, e.g., `MyBackend.cpp`, and placed in the directory

```
Backends/src/frontends
```

In either case, the function definition must be placed inside a designated namespace `Gambit::Backends::` *backend_name_safe_version*, automatically generated with the `BE_NAMESPACE` and `END_BE_NAMESPACE` macros.

```
BE_NAMESPACE
{
  Matrix<double,2,2> getMatrix()
  {
    // Call getMatrixElement four times
    // and return a complete matrix.
  }
}
END_BE_NAMESPACE
```

All backend functions and variables registered with the `BE_FUNCTION` and `BE_VARIABLE` macros (in the same frontend) can be accessed directly in convenience functions, as long as the body of the convenience function appears after their declarations. This also applies to calling convenience functions from each other.

Just like backend variables and regular backend functions, backend convenience functions can be declared as model-dependent, e.g.

```
BE_CONV_FUNCTION(getMatrix, Matrix<double,2,2>,
 (), "full_rotation_matrix", (model_α,
 model_β, ...) )
```

4.3 Backend initialisation functions

A backend library will usually have to be initialised in some way before any calculations can be performed. For instance, variables storing masses and couplings may have to be reset for every new parameter point. For this purpose, the user can define a backend initialisation function. This is a special kind of convenience function that automatically runs prior to any other backend operations. An initialisation function is registered by enclosing it within `BE_INI_FUNCTION` and `END_BE_INI_FUNCTION`. These macros automatically set up a void function taking no input arguments, so the user only has to supply the function body. As for backend convenience functions, this function definition can be placed either in the frontend header file or in the corresponding source file.

```
BE_INI_FUNCTION
{
  // Point-level initialisation.
}
END_BE_INI_FUNCTION
```

If some part of the initialisation only has to happen once for an entire scan, this can be accomplished by using a static flag:

```
BE_INI_FUNCTION
{
  static bool scan_level = true;
  if(scan_level)
  {
    // Scan-level initialisation.
  }
  scan_level = false;

  // Point-level initialisation.
}
END_BE_INI_FUNCTION
```

As with convenience functions, all registered backend functions and variables from the same backend are directly accessible from within the body of initialisation functions, so long as the body appears after the functions and variables have been declared.

To help with scan-level initialisation, GAMBIT provides a flag for every registered backend function, variable and convenience function, indicating whether or not it will be used in the upcoming scan. These flags are accessible only from a backend's initialisation function. The flags consist of pointers to boolean variables placed in the `InUse` namespace, i.e.

```
bool *InUse::name
```

where *name* is the name of the backend function, variable or convenience function as declared in the frontend header. Some example usage of the backend function `InUse` flags can be found in the fronted source files for nulike [15,53] and DDCalc [110].

Some backends write temporary files to disk during scan-level initialisation, which means that they cannot be safely initialised simultaneously in different MPI processes.[8] For such cases we provide a simple locking utility (`Utils::FileLock`) that can be employed to force serial execution of any block of code; example usage can be seen in the frontends to HiggsBounds and HiggsSignals [128–130].

In fact, backend initialisation functions are actually promoted to module function status, and associated with a special GAMBIT-internal module called BackendIniBit. This is how GAMBIT ensures that they always run before any other functions from their backend are used. This also allows backend initialisation functions to depend on input from other GAMBIT module functions. This is declared using the `BE_INI_DEPENDENCY` and `BE_INI_CONDITIONAL_DEPENDENCY` macros. These follow exactly the same syntax as the `DEPENDENCY` and `MODEL_CONDITIONAL_DEPENDENCY` macros for module functions (Sect. 3.1.2):

---

[8] This is also to be discouraged on basic efficiency grounds.

```
BE_INI_DEPENDENCY(capability, type)
BE_INI_CONDITIONAL_DEPENDENCY(capability, type, model_
  α, model_β, ...)
```

Thus, a backend initialisation function that needs to know the particle spectrum for the given parameter point could declare a dependency similar to

```
BE_INI_DEPENDENCY(particle_spectrum, Spectrum)
```

This will be fulfilled if some module function can provide the capability `particle_spectrum` of type `Spectrum`. The dependency can then be accessed from within the function body of the initialisation function,

```
const Spectrum& my_spec = *Dep::particle_spectrum;
```

This is similar to the way module functions access their dependencies (Sect. 3.2.1), except that for backend initialisation functions there is no need to specify the namespace `Pipes`::*function_name*.

4.4 Backend types

Backend functions and variables will often require types that are not known to GAMBIT, and which therefore need to be defined. For C and Fortran backends, these types are typically structs or typedefs involving only built-in C types. In this case the required definitions can be placed directly in a designated backend types header, named after the backend and placed in

```
Backends/include/gambit/Backends/backend_types
```

The types must live within the `Gambit` namespace, e.g.,

```
namespace Gambit
{
  struct Triplet
  {
    double x, y, z;
  };
}
```

but additional sub-namespaces can of course be used.

To ease the process of generating these type declarations and the `BE_FUNCTION` and `BE_VARIABLE` declarations that use them, GAMBIT ships with a simple utility for parsing Fortran backend code: CBGB, the Common Block harvester for GAMBIT Backends. CBGB automatically generates GAMBIT code that declares the necessary backend types, functions and variables, according to the list of functions and common blocks that a user chooses to interface with GAMBIT. CBGB is written in Python and can be found in `Backends/scripts/CBGB`.

CBGB takes a single configuration file as input. This file is written in Python syntax and must be placed in

`Backends/scripts/CBGB/configs`. An annotated example detailing all options and variables can be found in

```
Backends/scripts/CBGB/configs/example.py.
```

The most important variables to set in the configuration file are the three lists `input_files`, `load_functions` and `load_common_blocks`. We illustrate their use with a simple example, assuming a Fortran backend FortranBE v1.1:

```
input_files =
  ["../../installed/FortranBE/1.1/src/main.f
  90"]
load_functions = ["f1", "f2"]
load_common_blocks = ["cb"]
```

Here CBGB would parse the Fortran file `main.f90` and generate the `BE_FUNCTION` declarations needed to load the functions/subroutines `f1` and `f2`, as well as the type and `BE_VARIABLE` declarations required to load the common block `cb`. The file paths in `input_files` must either be absolute paths or relative to the `Backends/scripts/CBGB` directory. To ensure that the library symbol names used in `BE_FUNCTION` and `BE_VARIABLE` match those in the backend shared library, CBGB must also know which name mangling scheme to use. This is specified via the variable `name_mangling`, which can be set to either `"gfortran"`, `"ifort"` or `"g77"`.

Once the configuration file is ready, CBGB can be run by passing in this file as a command line argument, e.g.

```
python cbgb.py configs/FortranBE.py
```

The generated GAMBIT code is stored in the output files `backend_types_code.hpp` and `frontend_code.hpp`. In this example, the code in `backend_types_code.hpp` should be used in the backend types header `Backends/include/gambit/Backends/backend_types/FortranBE_1_1.hpp`, while the code in `frontend_code.hpp` should go in the frontend header `Backends/include/gambit/Backends/frontends/FortranBE_1_1.hpp`.

As GAMBIT itself is written in C++, interfacing with a Fortran backend requires translation between the Fortran types used in the backend and the corresponding C-family types. Therefore, GAMBIT provides several Fortran-equivalent types and typedefs for use in communicating with Fortran backends, with names indicating which Fortran type they correspond to:

```
Flogical
Flogical1
Finteger
Finteger2
Finteger4
Finteger8
Freal
Freal4
Freal8
Freal16
```

```
Fdouble
Fdoubleprecision
Fcomplex
Fcomplex8
Fcomplex16
Fdouble_complex
Flongdouble_complex
Fcharacter
```

These are the types that CBGB makes use of in the generated GAMBIT code. In cases where CBGB fails to correctly parse the Fortran code, the user must manually specify type, `BE_VARIABLE` and `BE_FUNCTION` declarations using the above Fortran-equivalent types.

There are important differences in how arrays are treated in Fortran compared to C/C++. First, the lower array index in Fortran is by default 1, in contrast to C/C++ arrays, which count from 0. More generally, Fortran allows the user to specify arbitrary index ranges, something that is not allowed in C/C++. In the case of multidimensional arrays, C/C++ arrays are stored in memory in row-major order, whereas Fortran arrays use column-major ordering, and the two types of arrays are therefore effectively transposed with respect to each other. To save the user from having to deal with these complexities, GAMBIT provides an `Farray` class for working with Fortran arrays. This class provides basic Fortran array semantics directly in C++ code. The class is templated on the array type and index ranges. Thus, a two-dimensional integer array with index ranges 1–3 and 1–4 can be declared as

```
Farray<Finteger,1,3,1,4> my_f_array;
```

We also provide a special `Fstring` class for working with Fortran strings. It takes the string length as a template argument

```
Fstring<4> my_f_string;
```

Similar to regular Fortran strings, any string longer than the specified length will be truncated, and shorter strings will be padded with trailing spaces.

More information about the GAMBIT Fortran compatibility types can be found in the in-code GAMBIT documentation (cf. Sect. 10.7), and in `Utils/include/gambit/Utils/util_types.hpp`.

4.5 Loading C++ classes at runtime with BOSS

Most physics tools written in C or Fortran are fundamentally just collections of functions and variables of standard types. In contrast, C++ tools typically define a number of new classes for the user to work with. Unfortunately, there exists no standard way of loading an *arbitrary* C++ class from a shared library at runtime. The `dl` library, itself written in C, only provides access to functions and global variables. This limitation can be overcome if the main application

has a predefined class interface that classes in the shared library are forced to adhere to; this is the so-called 'factory' pattern. This is unproblematic as long as all plug-ins are developed after the main application, which is normally the case. In GAMBIT, however, we face the reverse problem of turning already existing C++ physics tools into plug-ins for GAMBIT. To solve this problem we have developed the Python-based Backend-On-a-Stick Script (**BOSS**), which we describe here.

Strategies for runtime loading of classes are essentially always based on the C++ concept of *polymorphism*. One constructs a class interface from a base class containing a set of *virtual* member functions. These are functions for which the signature is defined, but where the actual implementation is expected to be overridden by classes that inherit from the base class. The idea can be illustrated by considering a base class `Polygon` containing a virtual member function `calculateArea`. From this base class two derived classes `Triangle` and `Rectangle` can be defined. Both classes should contain a `calculateArea` member function, but their implementations of this function would differ.

In plug-in, i.e. factory-based, systems, the main application defines the base class, while the plug-ins provide the specialized classes deriving from the base class. The main application can then be designed with the assumption that any future class passed in from a plug-in will have the predefined set of member functions, whose implementations live in the shared library that is loaded at runtime. The shared library also contains *factory functions*, one for each class it provides. These are functions that return a pointer to a newly created instance of a plug-in class. When a new class instance is required, the main application calls the correct factory function and interprets the pointer it receives as pointing to an instance of the known base class.

The purpose of BOSS is to reverse-engineer such a plug-in system for every backend class that is to be used from GAMBIT. Starting from a class `X` defined in the backend library, BOSS must generate source code for a base class with matching pure virtual member functions, as well as code for factory functions corresponding to the constructors `X(...)`. The generated base class is called `Abstract_X`, as classes containing pure virtual member functions are generally referred to as *abstract classes*. The source code for `Abstract_X` is added to both the backend source code and to GAMBIT. On the backend side, some additional source code is also inserted in the original class `X`, most importantly adding `Abstract_X` to the inheritance list of `X`. If class `X` originally inherits from a parent class `Y`, the abstract classes generated by BOSS mirror this structure. The resulting 'ladder pattern' is illustrated in Fig. 3.

When the ladder structure is complete, the basic ingredients for a plug-in system are in place. However, from the user perspective there are several limitations and inconveniences
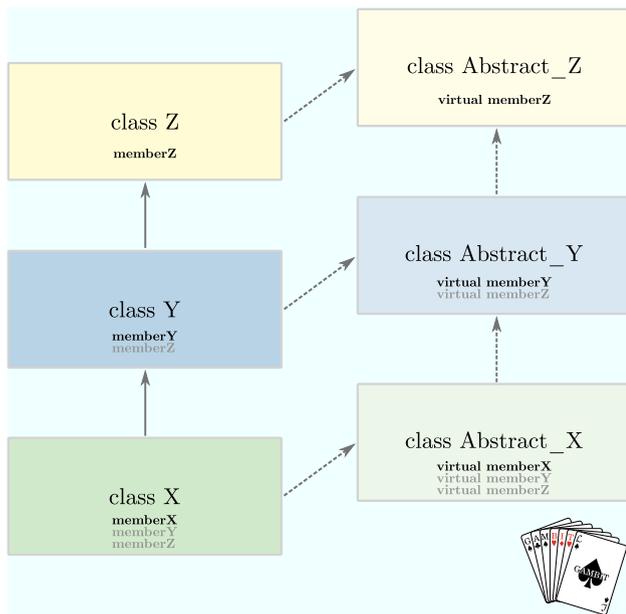
Fig. 3 The basic class 'ladder' pattern generated by **BOSS** in order to allow runtime loading of classes X, Y and Z, where Z is the parent of Y, which is in turn the parent of X. The original class hierarchy is mirrored by the abstract parent classes generated by **BOSS**. Virtual inheritance, illustrated here by dashed arrows, is used to avoid ambiguities. Member functions in the original classes are matched by pure virtual member functions in the abstract classes

inherrent in such a minimal system. For example, the factory functions must be called to create class instances, and class member variables cannot be accessed directly. To overcome such limitations, **BOSS** generates an additional layer in the form of an *interface class*, which mimics the user interface of the original class. It is this interface class that a user interacts with from within **GAMBIT**. The generated class is placed in a namespace constructed from the backend name and version, so if our example class X is part of **MyBackend** v1.2 the full name of the interface class will be `MyBackend_1_2::X`. However, from within a **GAMBIT** module function using this class, the shorter name X can be used.

Fundamentally, the interface class is just a wrapper for a pointer to the abstract class. Through a factory function, this pointer is initialised to point to an instance of the orginal class, thus establishing the connection between **GAMBIT** and the original class living in the backend library. In the example considered above, the class `MyBackend_1_2::X` would hold a pointer of type `Abstract_X`, pointing to an instance of X. This system is illustrated in Fig. 4. Note that the source code for the interface class is also inserted into the backend library. This allows **BOSS** to generate wrapper functions for any global library functions where the original class appears in the declaration.

When a **GAMBIT** module function requires classes from a backend library, this must be specified in the function's rollcall header entry by adding the macro
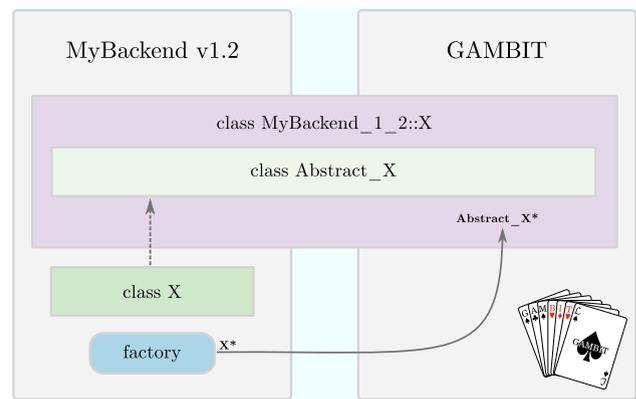
Fig. 4 Schematic diagram of the plug-in system generated by **BOSS** for the case where a backend library **MyBackend** 1.2 contains a single class X. For every constructor in X, a factory function returning a pointer to a new X instance is added to the library. An abstract base class `Abstract_X` and an interface class `MyBackend_1_2::X` are generated and added to both the backend library and **GAMBIT**. The interface class `MyBackend_1_2::X` wraps a pointer to `Abstract_X`. The factory function initialises this pointer with an instance of X, allowing **GAMBIT** to communicate with the original library class

```
NEEDS_CLASSES_FROM(backend_name, versions)
```

Here *versions* is an optional comma-separated list of permitted backend version numbers. If *versions* is left out or set to `default`, **GAMBIT** will use the default backend version specified in the header file `Backends/include/ gambit/ Backends/default_bossed_versions.hpp`. Here a default version can be chosen by setting a precompiler variable `Default_backend_name` to the desired safe version number, e.g.

```
#define Default_MyBackend 1_2
```

BOSS itself is the stand-alone **Python** program

```
Backends/scripts/BOSS/boss.py
```

For parsing the backend library source code **BOSS** employs the open-source tool **CastXML**.[9] The basic input to **BOSS** is a configuration file, written in **Python**, containing information about the backend library code that is to be 'BOSSed'. The configuration file should be placed in the `configs` subdirectory of the main **BOSS** directory. Here we will briefly go through the most important parts of the configuration file. For a complete list of options and variables we refer the reader to the example

```
Backends/scripts/BOSS/configs/Example_1_234.
  py
```

First the name and version number that **GAMBIT** should associate with the BOSSed library is set via the two variables `gambit_backend_name` and `gambit_backend_version`.

---

[9] http://github.com/CastXML/CastXML.

```
gambit_backend_name     = "MyBackend"
gambit_backend_version  = "1.2"
```

Then follows a set of path variables. All paths must be given either as absolute paths or relative to the main **BOSS** directory. Consider the following example:

```
input_files =
    ["../../installed/MyBackend/1.2/include/
     X.hpp"]
include_paths =
    ["../../installed/MyBackend/1.2/include"]
base_paths = ["../../installed/MyBackend/1.2"]
```

Here we assume that our example backend MyBackend 1.2 is located in

```
Backends/installed/MyBackend/1.2
```

The `input_files` variable is a list of the header files that contain the declarations for the classes and functions that are to be used from GAMBIT. Next, `include_paths` lists the paths where CastXML should search for any header files that are included from one of the input files. Finally, `base_paths` is a list of the base directories of the backend library. This is used by **BOSS** to differentiate between classes that are native to the backend and classes that are pulled in from external libraries.

BOSS generates a number of header and source files that must be included when the BOSSed backend is compiled into a shared library. The output paths for these files are set with the variables `header_files_to` and `src_files_to`, for instance

```
header_files_to =
    "../../installed/MyBackend/1.2/include"
src_files_to = "../../installed/MyBackend/1.2/src"
```

The next two variables, `load_classes` and `load_functions`, are lists containing the fully qualified names of the classes and functions to be loaded for use in GAMBIT. If we assume that in addition to the class `X`, MyBackend also contains a global function `addX` for adding two instances of `X`, we may have

```
load_classes = ["X"]
load_functions = ["addX(X, X)"]
```

Typically users will only need access to a subset of all the classes defined in the library, so only a subset of the available classes will be listed in `load_classes`. **BOSS** will then automatically limit the user interface of the BOSSed library to make sure that only functions and variables that make use of the loaded library classes and standard C++ types are accessible from GAMBIT. However, if the backend library includes some classes that are also independently included in GAMBIT, functions and variables relying on these classes should also be allowed as part of the BOSSed

library interface. Such classes can be listed in the dictionary `known_classes`. Here the dictionary key is the class name and the corresponding value is the header file where the class is declared.

BOSS is run by passing in the configuration file as a command line argument. For instance, with a configuration file `configs/MyBackend_1_2.py`, the command is simply

```
python boss.py configs/MyBackend_1_2.py
```

When **BOSS** finishes, a short set of instructions on how to connect the BOSSed library with GAMBIT is printed to `stdout`. Several of the variables in the configuration file can also be set directly as command line arguments to **BOSS**. For a complete list of arguments with explanations, see the output of the command

```
python boss.py --help
```

Although **BOSS** is able to provide runtime loading for most C++ classes and functions, there are some cases that the plug-in system generated by **BOSS** cannot handle yet. Most importantly, **BOSS** currently does not work with backend template classes, nor specialisations of C++ standard template library (STL) classes where the template parameter is a backend class. Further, the use of function pointers as function arguments or return types, and the use of C++11 features in function declarations, is only partially supported. When a limitation only affects a class member function or variable, **BOSS** will attempt to generate a limited class interface where the problematic element is excluded. Future versions of **BOSS** will improve on these limitations.

4.6 Backend information utility

Although most users will never have need to access it directly, we briefly point out here that a global backend information object exists in GAMBIT. It can be accessed by reference from any module function using the function `Backends::backendInfo()`. It provides a plethora of runtime information about which backends are presently connected, their versions, functions, classloading status and so on. The mostly likely use cases from within module functions for this object are to determine the folder in which a loaded backend resides:

```
std::string path_to_MyBackend_1_2 = Backends::
  backendInfo().path_dir("MyBackend", "1.2");
```

or to get the default version of a BOSSed backend required by an unversioned `NEEDS_CLASSES_FROM` declaration:

```
std::string default_MyBackend = Backends::
  backendInfo().default_version("MyBackend");
```

The full interface to this object can be found in `Backends/include/gambit/Backends/backend_info.hpp`.

5 Hierarchical model database

In GAMBIT, a **model** is defined to be a collection of named parameters. These parameters are intended to be sampled by some scanning algorithm, according to some chosen prior probability distribution.[10] The physical meaning of these parameters is defined entirely by how they are interpreted by **module functions**. It is up to the writers of modules to ensure that parameters are used in a consistent manner. Consistent usage is facilitated by the GAMBIT model database that the **dependency resolver** (Sect. 7) employs in order to automatically determine which module functions are compatible with which models. Module functions that are incompatible with the model(s) selected for scanning are disabled at runtime, and not considered during dependency resolution.

5.1 Model declaration

GAMBIT ships with a pre-defined selection of common models (Sect. 5.4). New models can be defined easily by adding an appropriate declaration in a new C++ header file located in the folder

```
Models/include/gambit/Models/models
```

When the GAMBIT build configuration is next re-run (see Sect. 11), the new model will be automatically detected and registered. The declarations of all the pre-defined models can also be found in this folder.

The syntax for declaring a simple two parameter model *my_model* with parameters *my_par1* and *my_par2* is:

```
#define MODEL my_model
START_MODEL
DEFINEPARS(my_par1, my_par2)
#undef MODEL
```

The `START_MODEL` command creates a `ModelParameters` object for the given model, which will hold the values of the parameters chosen at runtime by ScannerBit, and communicate them to relevant module functions during a scan. The macro `DEFINEPARS` is variadic, and can take up to 64 parameters (or more, depending on the user's version of Boost). If one prefers to break a long list of parameters into several pieces, this macro can be reused as many times as desired.

It is often the case that models will be subsets of a more general model, in the sense that a mapping from the general model to the more constrained model can be constructed. This hierarchical relationship between models is handled in GAMBIT by defining the general model to be a **parent** of the constrained model, with the constrained model being reciprocally defined as a **child** of that parent. The mapping from

---

[10] For frequentist sampling, the prior simply defines the distance measure on the parameter space to be used internally by the scanning algorithm when choosing new points.

the child parameters to the parent parameters is encoded in a translation function, which GAMBIT will call automatically when needed. Each parent model may have multiple children, however, a child model has only one parent. The "family tree" of any given model is thus a directed rooted tree graph, with the root of the tree being the common ancestor of all other models in the graph. The complete model database consists of a disconnected set of such family trees, see Fig. 2. When assessing the compatibility of module function with the model(s) being scanned, the GAMBIT dependency resolver automatically treats all module functions that declare compatibility with a given model as *also* compatible with all descendents of that model.

To declare that a model has a parent model *parent*, and assign a function *to_parent* capable of performing the translation from the child to the parent parameter set, the model declaration can be expanded to the following:

```
#define PARENT parent
  #define MODEL my_model
  START_MODEL
  DEFINEPARS(my_par1, my_par2)
  INTERPRET_AS_PARENT_FUNCTION(to_parent)
  #undef MODEL
#undef PARENT
```

If a model is declared to have a parent but no translation function, any attempt to use another function that depends on the translation will trigger a runtime error from the dependency resolver. Further details on declaring and defining translation functions can be found in Sect. 5.3. Note that we are only dealing with the abstract concept of translation functions between different model parameter spaces at this stage, not the actual physics of any translations in any given class of models. The actual translations between the models implemented in GAMBIT 1.0 are implied by the relations between parent and child models described in Sect. 5.4.

Putting these aspects together, complete model declarations can be very simple, as in the **CMSSM**:

```
// Must include models that are targets of
// translation functions
#include "gambit/Models/models/NUHM1.hpp"

#define MODEL CMSSM
#define PARENT NUHM1
  START_MODEL
  DEFINEPARS(M0,M12,A0,TanBeta,SignMu)
  INTERPRET_AS_PARENT_FUNCTION(CMSSM_to_NUHM1)
  // Translation functions defined in CMSSM.cpp
#undef PARENT
#undef MODEL
```

This declaration can be found in the model header `Models/include/gambit/Models/models/CMSSM.hpp`.

Directed cross-links between branches of a family tree, or even between trees, are also possible. Models related in this

way are denoted as **friend models**, though the relationship is not automatically mutual. If a model *my_model* has a friend model *friend*, then a function *to_friend* must also exist that can translate the parameters of *my_model* into those of *friend*. To declare such a relationship, one inserts the following into the model declaration for *my_model*:

```
INTERPRET_AS_X_FUNCTION (friend, to_friend)
```

With the addition of friend translations, the model hierarchy graph can become arbitrarily complicated. To avoid painful manual resolution of ambiguous translation pathways between models, it can be advisable to limit the number of friend links. Nevertheless, a large number of the possible ambiguities are automatically resolved by the default behaviour of the dependency resolver to prefer child-to-parent links over child-to-friend links. This behaviour can be disabled by switching the `prefer_model_specific_functions` option in the `KeyValues` section of the initialisation file to `false`. Manual resolution of all translation pathway ambiguities will then be required. Alternatively, one can simply add a `Rule` that overrides the default in a specific case. See Sect. 6 for details.

In some cases the translation from child to parent model, or to a friend model, may require the result of a calculation from a module function. It is therefore possible to declare **dependencies** for translation functions, which are directly analogous to the dependencies declared by module functions.[11]

In general, translation functions can depend on any other capability, which may be filled by functions from any module. The dependency resolution system ensures consistency of the requested and provided dependencies of all translation functions in such cases. For example, the translation functions might depend on some aspect of the particle spectrum, and may involve translation of parameters from one renormalisation scheme to another, from a UV-complete theory to an EFT, or from one renormalisation scale to another. In these examples, the dependencies would be most naturally resolved from **SpecBit**, if it possesses the relevant capability for the model in question; we refer readers to Ref. [111] for details of the functionalities available from this module.

To declare a dependency on some *capability* with C++ *type* for a child-to-parent translation function, one adds the following to the model declaration for the child:

```
INTERPRET_AS_PARENT_DEPENDENCY (capability, type)
```

To declare such a dependency for a translate-to-friend function, one instead adds:

```
INTERPRET_AS_X_DEPENDENCY (friend, capability, type)
```

---

[11] Internally, the translation functions actually *are* module functions, each belonging to a virtual module named after their source model.

where *friend* is the name of the target friend model. These declarations must appear after the declaration of the corresponding translation function.

The full machinery for declaring dependencies with complicated conditions on groups of models and backend choices – which is available for standard module functions – is not available for the dependencies of model translation functions. If this machinery is required, one should write a module function that uses it and returns a result associated with a new capability, which can then be made accessible in a translation function using the above declarations. In the most extreme case, this module function may perform the complete parameter translation and then simply store the results in a temporary `ModelParameters` object, which can then be retrieved by the "true" translation function via the above mechanism and simply copied into the target model `ModelParameters` object (see Sect. 5.3).

5.2 Model capabilities

In some cases a parameter in a **model** may directly correspond to a physically meaningful quantity. This quantity may be available already, computed in an alternate way, as the **capability** of some existing **module function**. One may wish to have the alternative of simply using the value of the parameter to satisfy the dependencies of other module functions on this quantity, rather than the module function calculation. It can therefore be convenient to directly inform GAMBIT of this correspondence when declaring a model. To declare this kind of relationship between a parameter *my_par* and a capability *capability*, one adds the following to the declaration of the model containing *my_par*:

```
MAP_TO_CAPABILITY (my_par, capability)
```

Of course the same could be achieved by manually creating a trivial module function that takes the model parameters as input, and then directly outputs one of them as its capability. Internally, `MAP_TO_CAPABILITY` causes GAMBIT to create a virtual module function of precisely this kind, but it is convenient to have this task performed automatically.

The module function so created has the same name as the parameter being mapped, and lives in the module corresponding to the model to which it belongs. Take the example of the top mass, a parameter of the **demo_A** model found (commented out) in `Models/include/gambit/Models/models/demo.hpp`:

```
MAP_TO_CAPABILITY (Mstop, Mstop_obs)
```

This declaration creates a new module function called `Mstop`, with capability `Mstop_obs` and return type `double`, and places it within the module named after **demo_A**. The function `demo_A::Mstop` simply returns the value of the `Mstop` parameter as it varies during a scan of **demo_A**.

This convenience facility exists for the simplest case only. In the case where the correspondence is not direct – for example, if a factor of two or a change of units is required, or if a dependency on some other calculation exists – then manually adding an additional module function to do the transformation is the only option.

5.3 Defining translation functions

In Sect. 5.1 we discussed how to declare a translation pathway between two models; we now turn to how to define the functions that actually perform the translation. These may or may not involve calculations relating to the spectrum (as in the example the referee is thinking of). , so in this case, they would depend on functions from SpecBit. The full details of how those functions work is provided in the SpecBit, DecayBit and PrecisionBit paper (1705.07936). In particular, this includes translations between pole and running masses in different schemes and EFTs.

The function definition can either be placed directly into the header file in which the source model is declared, or into a separate source file that includes the header. In the former case, the function body must appear *after* the `INTERPRET_AS` macro that declares it. In the latter case, the source file should be placed in

```
Models/src/models
```

to be properly auto-detected by the GAMBIT build system. Some extra headers providing additional helper macros should be included, and the names of the model and its parent redefined in order for the helpers to work properly. A basic template for such a file is:

```
#include "gambit/Models/model_macros.hpp"
#include "gambit/Models/model_helpers.hpp"
#include "gambit/Models/models/my_model.hpp"

#define MODEL     my_model
#define PARENT    parent

// function definition

#undef PARENT
#undef MODEL
```

Consider the following example function definition:

```
void MODEL_NAMESPACE::to_parent(const
 ModelParameters& myparams, ModelParameters&
 parentparams)
{
  double x = myparams["my_par"];
  parentparams.setValue("parent_par", 2*x);
}
```

Although this example is a child-to-parent translation function, the syntax is the same for child-to-friend functions. The translation function must return `void`, and take two arguments by reference: the source model parameters (which are `const`), and the target model parameters (of either the parent or friend model). The helper macro `MODEL_NAMESPACE` places the function in the correct namespace (`Gambit::Models::MODEL`), and relies on `MODEL` having been defined appropriately. On the first line of the function body, a parameter *my_par* is retrieved from the `ModelParameters` object, which contains the parameters of the source model. The value of this parameter is stored in the variable `x`. This is then multiplied by two, and used to set the value of the target model parameter *parent_par*, completing the parameter translation.

This example assumes that the target model has only one parameter, *parent_par*. Often a source and target model will have many overlapping parameters, and it is convenient to have a mechanism for copying all of these automatically, without modification. This can be done using the `setValues` member function of the target `ModelParameters` object:

```
parentparams.setValues(myparams, true);
```

The second parameter is optional, and `true` by default. This triggers an error if any of the parameters in `myparams` (from *my_model*) are missing from `parentparams` (from *parent*), i.e. if the source model parameter names are not a subset of the target model parameter names. Setting this `false` causes matching parameters to be copied but unmatched parameters to be ignored.

A real-world example that make use of `setValues` is the the **CMSSM**-to-**NUHM1** translation function (which was declared in Sect. 5.1):

```
#define MODEL CMSSM

  void MODEL_NAMESPACE::CMSSM_to_NUHM1
   (const ModelParameters &myP,
    ModelParameters &targetP)
  {
     logger()<<"Running interpret_as_parent "
            <<"calculations for CMSSM --> NUHM1."
            <<LogTags::info<<EOM;

     // Send all parameter values upstream
     // to matching parameters in parent.
     targetP.setValues(myP);

     // Set NUHM1 parameter mH equal to m0.
     targetP.setValue("mH", myP["M0"]);
  }

#undef MODEL
```

This function can be found in `Models/src/models/CMSSM.cpp`.

To retrieve dependencies on externally-calculated quantities, one uses regular module function syntax

```
USE_MODEL_PIPE(target)
const type* my_variable = *Dep::capability;
```

where the `USE_MODEL_PIPE` macro simply expands to a `using` statement that brings the **pipes** for the translation function into the current namespace, making the `Dep::`*capability* pointer easily accessible. The argument should be the name of the target (parent or friend) model, i.e. `USE_MODEL_PIPE`(*friend*) or `USE_MODEL_PIPE`(`PARENT`) (remembering that `PARENT` is a macro holding the actual parent model, defined in the model header).

5.4 Models defined in GAMBIT 1.0.0

Here we list the models already defined in the first release of GAMBIT, along with their parameters. The relationships between these models can be seen in Fig. 2.

5.4.1 Standard model

The SM exists in two parts within GAMBIT. The Higgs mass must be specified separately from the rest of the SM parameters, as it is often contained within the definition of BSM theories featuring BSM Higgs sectors. For those theories that do not include their own Higgs sector, e.g. **SingletDM**, we therefore provide additional models containing the Higgs mass as a parameter: **StandardModel_Higgs** and **StandardModel_Higgs_running**. Typically, one of these models should be scanned over in tandem with the rest of the SM (**StandardModel_SLHA2**) and the BSM theory in question. To investigate just the SM itself, one should perform a simultaneous scan of **StandardModel_SLHA2** and either **StandardModel_Higgs** or **StandardModel_ Higgs_running**.

StandardModel_SLHA2: `CKM_A`, `CKM_etabar`, `CKM_ lambda`, `CKM_rhobar`, `GF`, `alpha1`, `alpha2`, `alphaS`, `alphainv`, `delta13`, `mBmB`, `mCmC`, `mD`, `mE`, `mMu`, `mNu1`, `mNu2`, `mNu3`, `mS`, `mT`, `mTau`, `mU`, `mZ`, `theta12`, `theta13`, `theta23`.
    This model contains the SM parameters defined in the `SMINPUTS`, `VCKMIN` and `UPMNSIN` blocks of the second SUSY Les Houches Accord (SLHA2; [167]). This includes the Z pole mass, the Fermi (weak) coupling G_F, the strong and electromagnetic couplings at scale m_Z in the $\overline{MS}$ renormalisation scheme, pole masses for leptons, neutrinos and the top quark, running masses for other quarks in the $\overline{MS}$ scheme (at scale m_b for b, m_c for c and $2\,\text{GeV}$ for u, d and s), the CKM mixing matrix in Wolfenstein parameterisation, and the PMNS matrix, characterised by three mixing angles and three CP-violating phases. To convert the Wolfenstein parameters into V_{CKM} entries internally, we use the 9th-order

expansions of Ref. [32]. More detailed definitions of these parameters can be found in Appendix C.

StandardModel_Higgs_running: `QEWSB`, `mH`.
    This model provides a description of the SM Higgs sector in terms of m_H^2, the bare Higgs mass parameter in the SM Lagrangian at scale m_Z. The vacuum expectation value of the SM Higgs field at m_Z can be obtained from the **StandardModel_SLHA2** as $v_0 = (\sqrt{2}G_F)^{-1/2}$. This model is intended for use in situations where the Higgs potential is run to different scales, e.g. for calculating pole masses or investigating vacuum stability. It therefore also contains one additional parameter: Q_{EWSB}, the scale at which the electroweak symmetry-breaking (EWSB) consistency condition that the Higgs potential possess a tree-level minimum is imposed. Although in principle physical properties should not depend on its value, typically one prefers to take $Q_{\text{EWSB}} \sim m_t$ in order to minimise errors from neglecting higher-order loops. It is common to vary this parameter by a factor of a few in order to try to quantify the uncertainty in resulting pole masses from missing loop terms.

StandardModel_Higgs: `mH`.
    Unlike the **StandardModel_Higgs_running** model, the tree-level Higgs mass m_h is taken as the free parameter of **StandardModel_Higgs**, but interpreted directly as the pole mass for most calculations. This generally removes the need to calculate it via renormalisation group running in any higher-energy theory. For simple calculations, this allows a cut-down GAMBIT `Spectrum` object to be produced, with no ability to run, and the Higgs 'pole' mass extracted from it by simply accessing the value of the tree-level parameter for the given point in parameter space. When observables are to be calculated that genuinely need to use running parameters, the model point is up-translated to a parameter point in the **StandardModel_Higgs_running** (the parent model), where m_H^2 at scale m_Z is set equal to the square of the tree-level mass, and Q_{EWSB} is set to m_t. This is useful for more detailed calculations involving module functions that explicitly require the running mass parameter of **StandardModel_Higgs_ running**, and/or functions that need accurate pole masses calculated by including the Higgs sector in renormalisation calculations.

5.4.2 Scalar singlet

The scalar singlet is the simplest possible model for DM, consisting of a single additional scalar field S uncharged under the gauge symmetries of the SM, and stabilised by

a $\mathbb{Z}_2$ symmetry. The additional renormalisable Lagrangian terms permitted by general symmetry arguments are

$$\mathcal{L}_{SS} = \frac{1}{2}\mu_S^2 S^2 + \frac{1}{2}\lambda_{hs} S^2 |H|^2 + \frac{1}{4}\lambda_s S^4 + \frac{1}{2}\partial_\mu S \partial^\mu S. \quad (1)$$

From left to right, these are: the bare S mass, the dimension-4 Higgs-portal coupling, the S quartic self-coupling, and the kinetic term. The latter plays no role in phenomenology, leaving three free parameters of the theory: μ_S^2, λ_{hs} and λ_s. After EWSB, the singlet mass receives an additional contribution from the Higgs-portal term, leading to a tree-level mass of

$$m_S = \sqrt{\mu_S^2 + \frac{1}{2}\lambda_{hs} v_0^2}. \quad (2)$$

This model has been subjected to global fits in Refs. [99,107, 121].

SingletDM_running: `lambda_S, lambda_hS, mS`.
   This model has μ_S^2, λ_{hs} and λ_s at scale m_Z as free parameters, allowing full calculation of pole masses, renormalisation group running and vacuum stability.

SingletDM: `lambda_hS, mS`.
   The relationship between **SingletDM** and **SingletDM_running** is analogous to the one between **StandardModel_Higgs** and **StandardModel_Higgs_running**. SingletDM has m_S as a free parameter, leading to two use cases. The first is to interpret the model parameter directly as the pole mass for S and do phenomenology without any spectrum calculation; the second is to take the parameter m_S as the tree-level estimate of the mass, use Eq. 2 to recover μ_S^2 matched to the tree-level mass at scale m_Z, and calculate the pole mass accordingly in the parent model **SingletDM_running**. One chooses between these two options by selecting which function from **SpecBit** to obtain a `Spectrum` object from. SingletDM includes the Higgs-portal coupling λ_{hs} identically to the parent model, which we also set the running coupling at m_Z to when translating a **SingletDM** model point to a **SingletDM_running** point. It however does not include any description of the quartic coupling. This is because the S self-coupling term only plays a role in observables via RGE running, such as in the calculation of pole masses and analysis of Higgs vacuum stability. When translating to the parent model, we therefore explicitly choose the quartic coupling to be absent at scale m_Z (even though it will be regenerated at other scales under RGE flow):

$$\lambda_S(m_Z) = 0. \quad (3)$$

5.4.3 Weak-scale MSSM

These models feature MSSM soft SUSY-breaking Lagrangian parameters defined at a chosen scale Q, typically set to something near the weak scale.

The MSSM is the version of SUSY containing the least additional field content beyond the SM. Its Lagrangian (see e.g. [168])

$$\mathcal{L}_{MSSM} = \mathcal{L}_{SUSY\text{-}SM} + \mathcal{L}_{soft}, \quad (4)$$

is obtained by supersymmetrising the (pre-EWSB) SM Lagrangian to find $\mathcal{L}_{SUSY\text{-}SM}$, and augmenting it with all possible renormalisable soft SUSY-breaking terms that conserve both baryon (B) and lepton number (L). These soft terms are

$$\mathcal{L}_{soft} = -\frac{1}{2}\left[M_1 \bar{\tilde{B}}^0 \tilde{B}^0 + M_2 \bar{\tilde{W}}_A \tilde{W}_A + M_3 \bar{\tilde{g}}_B \tilde{g}_B \right] \quad (5a)$$

$$-\frac{i}{2}\left[M_1' \bar{\tilde{B}}^0 \gamma_5 \tilde{B}^0 + M_2' \bar{\tilde{W}}_A \gamma_5 \tilde{W}_A + M_3' \bar{\tilde{g}}_B \gamma_5 \tilde{g}_B \right] \quad (5b)$$

$$-\epsilon_{ab}\left[b H_u^a H_d^b + \text{h.c.} \right] - m_{H_u}^2 |H_u|^2 - m_{H_d}^2 |H_d|^2 \quad (5c)$$

$$+\sum_{i,j=1,3}\left\{ -\left[\tilde{Q}_i^\dagger (\mathbf{m}_Q^2)_{ij} \tilde{Q}_j + \tilde{d}_{Ri}^\dagger (\mathbf{m}_d^2)_{ij} \tilde{d}_{Rj} \right.\right.$$
$$\left. + \tilde{u}_{Ri}^\dagger (\mathbf{m}_u^2)_{ij} \tilde{u}_{Rj} + \tilde{L}_i^\dagger (\mathbf{m}_L^2)_{ij} \tilde{L}_j + \tilde{e}_{Ri}^\dagger (\mathbf{m}_e^2)_{ij} \tilde{e}_{Rj} \right] \quad (5d)$$

$$-\epsilon_{ab}\left[(\mathbf{T}_u)_{ij} \tilde{Q}_i^a H_u^b \tilde{u}_{Rj}^\dagger - (\mathbf{T}_d)_{ij} \tilde{Q}_i^a H_d^b \tilde{d}_{Rj}^\dagger \right.$$
$$\left. - (\mathbf{T}_e)_{ij} \tilde{L}_i^a H_d^b \tilde{e}_{Rj}^\dagger + \text{h.c.} \right] \quad (5e)$$

$$-\epsilon_{ab}\left[(\mathbf{C}_u)_{ij} \tilde{Q}_i^a H_d^{*b} \tilde{u}_{Rj}^\dagger - (\mathbf{C}_d)_{ij} \tilde{Q}_i^a H_u^{*b} \tilde{d}_{Rj}^\dagger \right.$$
$$\left.\left. - (\mathbf{C}_e)_{ij} \tilde{L}_i^a H_u^{*b} \tilde{e}_{Rj}^\dagger + \text{h.c.} \right]\right\}. \quad (5f)$$

Here we explicitly sum over the generation indices i and j, and imply summation over the gauge generator indices $A = 1 \cdots 3$ and $B = 1 \cdots 8$, as well as the $SU(2)_L$ indices $a, b = 1, 2$. Here ϵ_{ab} is the two-dimensional completely antisymmetric tensor, defined such that $\epsilon_{12} = -\epsilon_{21} = 1$. Superparticle fields are denoted by tilded operators ($\tilde{Q}_j$, $\tilde{u}_{Rj}^\dagger$, etc.), where $\tilde{B}^0$, $\tilde{W}_A$ and $\tilde{g}_B$ are the superpartners of the SM gauge bosons. Fields denoted with capital letters are $SU(2)_L$ doublets ($Q_i \equiv (u_{Li}, d_{Li})^T$, etc), whereas lowercase fields are $SU(2)_L$ singlets. The subscripts u and d refer to the two Higgs doublets, which give masses separately to up- and down-type quarks.

The first two terms in Eq. 5 (5a, 5b) are explicit gaugino masses associated with the real parameters M_1, M_2, M_3 and M_1', M_2', M_3'. The second set of these violates CP, so M_1', M_2' and M_3' should be very small to agree with exper-

iment. The Higgs sector (5c) includes explicit mass terms with real parameters $m_{H_u}^2$ and $m_{H_d}^2$, as well as a bilinear coupling with complex parameter b. Explicit sfermion masses (5d) come from the five 3×3 Hermitian mass-squared matrices $\mathbf{m}_Q^2$, $\mathbf{m}_u^2$, $\mathbf{m}_d^2$, $\mathbf{m}_L^2$ and $\mathbf{m}_e^2$. The final two terms (5e, 5f) denote trilinear couplings between the Higgs and squarks or sleptons, with general Yukawa-type complex 3×3 matrices $\mathbf{T}_u$, $\mathbf{T}_d$, $\mathbf{T}_e$ and $\mathbf{C}_u$, $\mathbf{C}_d$, $\mathbf{C}_e$. The $\mathbf{C}$ terms are often omitted from the definition of the MSSM, as they end up strongly suppressed in many SUSY-breaking schemes. SUSY-breaking scenarios often imply universality relations between the Yukawa-scaled soft-breaking trilinear couplings $\mathbf{A}_u$, $\mathbf{A}_d$ and $\mathbf{A}_e$, which are defined as

$$(\mathbf{A}_f)_{ij} \equiv (\mathbf{T}_f)_{ij}/(\mathbf{Y}_f)_{ij} \quad f \in \{u, d, e\}, \tag{6}$$

where i and j run over all three generations, but summation is not implied.

$\mathcal{L}_{\text{SUSY-SM}}$ contains derivatives of the superpotential

$$\hat{W} = \epsilon_{ab} \left\{ \sum_{i,j=1,3} \left[(\mathbf{Y}_u)_{ij} \hat{Q}_i^a \hat{H}_u^b \hat{U}_j^c - (\mathbf{Y}_d)_{ij} \hat{Q}_i^a \hat{H}_d^b \hat{D}_j^c \right. \right.$$
$$\left. \left. - (\mathbf{Y}_e)_{ij} \hat{L}_i^a \hat{H}_d^b \hat{E}_j^c \right] - \mu \hat{H}_u^a \hat{H}_d^b \right\}. \tag{7}$$

Here the indices i and j are again generation number, a and b are $SU(2)_L$ indices and ϵ_{ab} is the two-dimensional antisymmetric tensor. Carets indicate superfields. The terms $\hat{U}_j^c$, $\hat{D}_j^c$ and $\hat{E}_j^c$ are the left chiral superfields containing the charge conjugates of the right-handed $SU(2)_L$ singlets: up-type (s)quarks, down-type (s)quarks and (s)electrons, respectively. Derivatives of $\hat{W}$ with respect to its scalar fields give rise to all non-gauge interaction terms in $\mathcal{L}_{\text{SUSY-SM}}$. It plays a similar role to the non-gauge part of the scalar potential in non-supersymmetric theories, specifying the Higgs potential via the complex parameter μ and the Higgs-fermion interactions and fermion masses via the complex 3×3 Yukawa coupling matrices $\mathbf{Y}_u$, $\mathbf{Y}_d$ and $\mathbf{Y}_e$.

R parity is conserved in Eqs. 5 and 7 by construction, by virtue of B and L being conserved individually. This makes the lightest SUSY particle (LSP) absolutely stable; in general we impose the condition that this must be the lightest neutralino. Unless otherwise noted, we neglect the phenomenology of the gravitino, assuming it to be sufficiently heavy that it is not the LSP and its decays at early times are irrelevant.

MSSM63atQ: Ad_11, Ad_12, Ad_13, Ad_21, Ad_22, Ad_23, Ad_31, Ad_32, Ad_33, Ae_11, Ae_12, Ae_13, Ae_21, Ae_22, Ae_23, Ae_31, Ae_32, Ae_33, Au_11, Au_12, Au_13, Au_21, Au_22, Au_23, Au_31, Au_32, Au_33, M1, M2, M3, Qin, SignMu, TanBeta, mHd2, mHu2, md2_11, md2_12, md2_13, md2_22, md2_23, md2_33, me2_11, me2_12, me2_13, me2_22, me2_23, me2_33, ml2_11, ml2_12, ml2_13, ml2_22, ml2_23, ml2_33, mq2_

11, mq2_12, mq2_13, mq2_22, mq2_23, mq2_33, mu2_11, mu2_12, mu2_13, mu2_22, mu2_23, mu2_33.

This model contains 65 free parameters: the scale Q, the sign of the μ parameter, and 63 parameters of the MSSM Lagrangian. Because of their usual irrelevance in (known) SUSY-breaking schemes, here we set the $\mathbf{C}$ terms to zero. Apart from this omission, the MSSM63 is the most general formulation of the CP-conserving MSSM: here the CP-violating gaugino masses M' and complex phases are all also explicitly set to zero. This leaves 3 free gaugino masses M_1, M_2 and M_3, 6 real parameters each from the mass-squared matrices $\mathbf{m}_Q^2$, $\mathbf{m}_u^2$, $\mathbf{m}_d^2$, $\mathbf{m}_L^2$ and $\mathbf{m}_e^2$ and a further 9 each from the trilinear couplings $\mathbf{A}_u$, $\mathbf{A}_d$ and $\mathbf{A}_e$. The final three parameters come from the Higgs sector, where we have $m_{H_u}^2$ and $m_{H_d}^2$, and trade b and μ for the sign of μ and the ratio of the up-type to down-type Higgs vacuum expectation values $\tan\beta \equiv v_u/v_d$. All parameters are defined in the $\overline{DR}$ scheme at the scale Q, except for $\tan\beta$, which is defined at m_Z.

Relative to the general MSSM, the additional constraints applied in this model are:

$$M_1' = M_2' = M_3' = 0, \tag{8}$$
$$\mathbf{C}_u = \mathbf{C}_d = \mathbf{C}_e = \mathbf{0}, \tag{9}$$
$$\mathbf{m}_Q^2, \mathbf{m}_u^2, \mathbf{m}_d^2, \mathbf{m}_L^2, \mathbf{m}_e^2, \mathbf{A}_u, \mathbf{A}_d, \mathbf{A}_e \text{ all real.} \tag{10}$$

MSSM30atQ: Ad_1, Ad_2, Ad_3, Ae_1, Ae_2, Ae_3, Au_1, Au_2, Au_3, M1, M2, M3, Qin, SignMu, TanBeta, mHd2, mHu2, md2_1, md2_2, md2_3, me2_1, me2_2, me2_3, ml2_1, ml2_2, ml2_3, mq2_1, mq2_2, mq2_3, mu2_1, mu2_2, mu2_3.
As per the **MSSM63atQ**, but with all off-diagonal elements in $\mathbf{m}_Q^2$, $\mathbf{m}_u^2$, $\mathbf{m}_d^2$, $\mathbf{m}_L^2$, $\mathbf{m}_e^2$, $\mathbf{A}_u$, $\mathbf{A}_d$ and $\mathbf{A}_e$ set to zero, in order to suppress flavour-changing neutral currents:

$$\mathbf{m}_Q^2, \mathbf{m}_u^2, \mathbf{m}_d^2, \mathbf{m}_L^2, \mathbf{m}_e^2, \mathbf{A}_u, \mathbf{A}_d, \mathbf{A}_e \text{ diagonal.} \tag{11}$$

MSSM25atQ: Ad_3, Ae_12, Ae_3, Au_3, M1, M2, M3, Qin, SignMu, TanBeta, mHd2, mHu2, md2_1, md2_2, md2_3, me2_1, me2_2, me2_3, ml2_1, ml2_2, ml2_3, mq2_1, mq2_2, mq2_3, mu2_1, mu2_2, mu2_3.
This was the model investigated in Ref. [169]. As per the **MSSM30atQ**, but with first and second-generation trilinear couplings degenerate in the slepton sector, and set to zero for squarks:

$$(\mathbf{A}_e)_{11} = (\mathbf{A}_e)_{22}, \tag{12}$$
$$(\mathbf{A}_u)_{11} = (\mathbf{A}_u)_{22} = (\mathbf{A}_d)_{11} = (\mathbf{A}_d)_{22} = 0. \tag{13}$$

MSSM24atQ: `Ad_3, Ae_3, Au_3, M1, M2, M3, Qin, SignMu, TanBeta, mHd2, mHu2, md2_1, md2_2, md2_3, me2_1, me2_2, me2_3, ml2_1, ml2_2, ml2_3, mq2_1, mq2_2, mq2_3, mu2_1, mu2_2, mu2_3.`
As per the **MSSM25atQ**, but with first and second-generation trilinear couplings in the slepton sector also set to zero:

$$(\mathbf{A}_u)_{ii} = (\mathbf{A}_d)_{ii} = (\mathbf{A}_e)_{ii} = 0 \quad \forall i \in \{1, 2\}. \tag{14}$$

MSSM20atQ: `Ad_3, Ae_12, Ae_3, Au_3, M1, M2, M3, Qin, SignMu, TanBeta, mHd2, mHu2, md2_12, md2_3, me2_12, me2_3, ml2_12, ml2_3, mq2_12, mq2_3, mu2_12, mu2_3.`
As per the **MSSM25atQ**, but with degenerate first and second-generation sfermion mass parameters:

$$(\mathbf{A}_e)_{11} = (\mathbf{A}_e)_{22}, \tag{15}$$
$$(\mathbf{A}_u)_{11} = (\mathbf{A}_u)_{22} = (\mathbf{A}_d)_{11} = (\mathbf{A}_d)_{22} = 0, \tag{16}$$
$$(\mathbf{m}_X^2)_{11} = (\mathbf{m}_X^2)_{22} \quad \forall X \in \{Q, u, d, L, e\}. \tag{17}$$

MSSM19atQ: `Ad_3, Ae_3, Au_3, M1, M2, M3, Qin, SignMu, TanBeta, mHd2, mHu2, md2_12, md2_3, me2_12, me2_3, ml2_12, ml2_3, mq2_12, mq2_3, mu2_12, mu2_3.`
This is the model that is sometimes referred to as the "phenomenological" MSSM (pMSSM). It has been the focus of many non-statistical random parameter scans, e.g. [2,118,170–172]. As per the **MSSM20atQ**, but with first and second-generation trilinear couplings in the slepton sector also set to zero:

$$(\mathbf{A}_u)_{ii} = (\mathbf{A}_d)_{ii} = (\mathbf{A}_e)_{ii} = 0 \quad \forall i \in \{1, 2\}, \tag{18}$$
$$(\mathbf{m}_X^2)_{11} = (\mathbf{m}_X^2)_{22} \quad \forall X \in \{Q, u, d, L, e\}. \tag{19}$$

MSSM16atQ: `Ad_3, Ae_3, Au_3, M1, M2, M3, Qin, SignMu, TanBeta, mHd2, mHu2, md2_3, me2_3, ml2_12, ml2_3, mq2_12, mq2_3, mu2_3.`
As per the **MSSM19atQ**, but with all first and second generation squark mass parameters degenerate, and all first and second generation slepton mass parameters degenerate:

$$(\mathbf{m}_Q^2)_{ii} = (\mathbf{m}_u^2)_{jj} = (\mathbf{m}_d^2)_{kk} \quad \forall i, j, k \in \{1, 2\}, \tag{20}$$
$$(\mathbf{m}_L^2)_{ii} = (\mathbf{m}_e^2)_{jj} \quad \forall i, j \in \{1, 2\}. \tag{21}$$

MSSM15atQ: `A0, At, M1, M2, M3, Qin, SignMu, TanBeta, mHd2, mHu2, md2_3, me2_3, ml2_12, ml2_3 mq2_12, mq2_3, mu2_3.`
This is the model explored in Ref. [89], up to reparameterisation of the Higgs sector. As per the **MSSM16atQ**, but

with down-type and sleptonic trilinear couplings degenerate:

$$(\mathbf{A}_d)_{33} = (\mathbf{A}_e)_{33}. \tag{22}$$

MSSM11atQ: `Ad_3, Ae_3, Au_3, M1, M2, M3, Qin, SignMu, TanBeta, mHd2, mHu2, ml2, mq2.`
As per the **MSSM16atQ**/**MSSM19atQ**, but with universal squark ($m_{\tilde{q}}^2$) and slepton ($m_{\tilde{l}}^2$) mass parameters:

$$(\mathbf{m}_X^2)_{ii} \equiv m_{\tilde{q}}^2 \quad \forall i \in \{1 \cdots 3\}, X \in \{Q, u, d\}, \tag{23}$$
$$(\mathbf{m}_Y^2)_{ii} \equiv m_{\tilde{l}}^2 \quad \forall i \in \{1 \cdots 3\}, Y \in \{L, e\}. \tag{24}$$

MSSM10atQ: `Ad_3, Au_3, M1, M2, M3, Qin, SignMu, TanBeta, mHd2, mHu2, ml2, mq2.`
As per the **MSSM11atQ**, but with no sleptonic trilinear coupings:

$$(\mathbf{A}_e)_{33} = 0. \tag{25}$$

MSSM10batQ: `Ad_3, Ae_3, Au_3, M1, M2, M3, Qin, SignMu, TanBeta, mHd2, mHu2, mf2.`
As per the **MSSM11atQ**, but with a universal sfermion mass parameter $m_{\tilde{f}}^2$:

$$m_{\tilde{q}}^2 = m_{\tilde{l}}^2 \equiv m_{\tilde{f}}^2. \tag{26}$$

MSSM10catQ: `A0, M1, M2, M3, Qin, SignMu, Tan Beta, mHd2, mHu2, ml2, mq2_12, mq2_3.`
This is the model explored in Ref. [92], up to reparameterisation of the Higgs sector. As per the **MSSM15atQ**, but with a universal trilinear coupling A_0, 3rd generation squark mass ($m_{\tilde{q}3}^2$) and slepton mass ($m_{\tilde{l}}^2$) parameters:

$$(\mathbf{A}_u)_{33} = (\mathbf{A}_d)_{33} = (\mathbf{A}_e)_{33} \equiv A_0, \tag{27}$$
$$(\mathbf{m}_Q^2)_{33} = (\mathbf{m}_u^2)_{33} = (\mathbf{m}_d^2)_{33} \equiv m_{\tilde{q}3}^2, \tag{28}$$
$$(\mathbf{m}_L^2)_{ii} = (\mathbf{m}_e^2)_{jj} \equiv m_{\tilde{l}}^2 \quad \forall i, j \in \{1 \cdots 3\}. \tag{29}$$

MSSM9atQ: `Ad_3, Au_3, M1, M2, M3, Qin, SignMu, TanBeta, mHd2, mHu2, mf2` As per the **MSSM11atQ**, but with both the approximations introduced in the **MSSM10atQ** and **MSSM10batQ**, i.e. universal sfermion masses and no sleptonic trilinear couplings:

$$(\mathbf{A}_e)_{33} = 0, \tag{30}$$
$$m_{\tilde{q}}^2 = m_{\tilde{l}}^2 \equiv m_{\tilde{f}}^2. \tag{31}$$

MSSM7atQ: `Ad_3, Au_3, M2, Qin, SignMu, TanBeta, mHd2, mHu2, mf2.`
This model has been used extensively in DarkSUSY papers, e.g. [147,173,174]. As per the **MSSM9atQ**, but

assuming a Grand Unified Theory (GUT)-inspired relationship between the gaugino masses.

$$\frac{3}{5}\cos^2\theta_W M_1 = \sin^2\theta_W M_2 = \frac{\alpha}{\alpha_s}M_3. \tag{32}$$

When implementing this relationship, we use $\sin^2\theta_W$ at the Z pole mass scale, which we calculate directly from the **StandardModel_SLHA2** parameters G_F, m_Z (pole) and $\alpha_{\overline{MS}}^{-1}(m_Z)$.

5.4.4 GUT-scale MSSM

These models feature MSSM soft SUSY-breaking Lagrangian parameters defined at the scale of gauge coupling unification, typically referred to as the GUT scale.

MSSM63atMGUT: Ad_11, Ad_12, Ad_13, Ad_21, Ad_22, Ad_23, Ad_31, Ad_32, Ad_33, Ae_11, Ae_12, Ae_13, Ae_21, Ae_22, Ae_23, Ae_31, Ae_32, Ae_33, Au_11, Au_12, Au_13, Au_21, Au_22, Au_23, Au_31, Au_32, Au_33, M1, M2, M3, SignMu, TanBeta, mHd2, mHu2, md2_11, md2_12, md2_13, md2_22, md2_23, md2_33, me2_11, me2_12, me2_13, me2_22, me2_23, me2_33, ml2_11, ml2_12, ml2_13, ml2_22, ml2_23, ml2_33, mq2_11, mq2_12, mq2_13, mq2_22, mq2_23, mq2_33, mu2_11, mu2_12, mu2_13, mu2_22, mu2_23, mu2_33.
As per the **MSSM63atQ**, but with Q set to the GUT scale. Translation to **MSSM63atQ** requires having already solved the renormalisation group equations (RGEs) for the model, in order to determine the value of the GUT scale.

MSSM30atMGUT: Ad_1, Ad_2, Ad_3, Ae_1, Ae_2, Ae_3, Au_1, Au_2, Au_3, M1, M2, M3, SignMu, TanBeta, mHd2, mHu2, md2_1, md2_2, md2_3, me2_1, me2_2, me2_3, ml2_1, ml2_2, ml2_3, mq2_1, mq2_2, mq2_3, mu2_1, mu2_2, mu2_3.
This is the **MSSM30atQ** with $Q = M_{GUT}$; as per the **MSSM63atMGUT**, but with all off-diagonal elements in $\mathbf{m}_Q^2$, $\mathbf{m}_u^2$, $\mathbf{m}_d^2$, $\mathbf{m}_L^2$, $\mathbf{m}_e^2$, $\mathbf{A}_u$, $\mathbf{A}_d$ and $\mathbf{A}_e$ set to zero, in order to suppress flavour-changing neutral currents:

$$\mathbf{m}_Q^2, \mathbf{m}_u^2, \mathbf{m}_d^2, \mathbf{m}_L^2, \mathbf{m}_e^2, \mathbf{A}_u, \mathbf{A}_d, \mathbf{A}_e \text{ diagonal.} \tag{33}$$

NUHM2: A0, M0, M12, SignMu, TanBeta, mHd, mHu.
The second Non-Universal Higgs Mass model. Descended from the **MSSM63atMGUT**. All off-diagonal elements in $\mathbf{m}_Q^2$, $\mathbf{m}_u^2$, $\mathbf{m}_d^2$, $\mathbf{m}_L^2$ and $\mathbf{m}_e^2$ are set to zero, and all diagonal elements are set equal to a universal sfermion mass m_0. All gaugino masses are set to the universal mass $m_{1/2}$, and all entries in $\mathbf{A}_u$, $\mathbf{A}_d$ and $\mathbf{A}_e$ are set to a universal trilinear coupling A_0. Global fits of this model have been

performed in Refs. [69,85,119].

$$\mathbf{m}_Q^2, \mathbf{m}_u^2, \mathbf{m}_d^2, \mathbf{m}_L^2, \mathbf{m}_e^2 \text{ diagonal,} \tag{34}$$

$$M_1 = M_2 = M_3 \equiv m_{1/2}, \tag{35}$$

$$(\mathbf{m}_X^2)_{ii} \equiv m_0^2 \ \forall \ i \in \{1\cdots3\}, X \in \{Q, u, d, L, e\}, \tag{36}$$

$$(\mathbf{A}_Y)_{ij} \equiv A_0 \ \forall \ i, j \in \{1\cdots3\}, Y \in \{u, d, e\}. \tag{37}$$

NUHM1: A0, M0, M12, SignMu, TanBeta, mH.
The first Non-Universal Higgs Mass model, fitted in Refs. [54,61,67,68,119]. As per the **NUHM2**, but with a single Higgs mass parameter m_H:

$$m_{H_u}^2 = m_{H_d}^2 \equiv (m_H)^2. \tag{38}$$

CMSSM: A0, M0, M12, SignMu, TanBeta.
The Constrained MSSM, most notably fitted in recent years in Refs. [68,88,90,119]. As per the **NUHM1**, but with m_0 playing the role of a fully universal scalar mass parameter:

$$m_H = m_0. \tag{39}$$

mSUGRA: A0, M0, M12, SignMu, TanBeta.
The most common definition of the minimal supergravity model; just a pseudonym for the **CMSSM**.[12]

5.4.5 Flavour EFT

The study of rare meson decays is typically done within the framework of effective field theory (EFT), where squared matrix elements for decays from initial states i to final states f are calculated from $|\langle f|\mathcal{H}_{eff}|i\rangle|^2$, using an interaction Hamiltonian

$$\mathcal{H}_{eff} = -\frac{4G_F}{\sqrt{2}} V_{tb} V_{ts}^* \sum_x C_x(\mu)\mathcal{O}_x(\mu). \tag{40}$$

Here μ specifies the scale of the process, V is the CKM matrix and G_F is the Fermi constant. $\mathcal{H}_{eff}$ is decomposed into a linear combination of effective interactions $\mathcal{O}_x$ with Wilson coefficients C_x. Some such interactions exist already in the SM, e.g.

$$\begin{aligned}
\mathcal{O}_7 &= \frac{e}{(4\pi)^2}m_b(\bar{s}\sigma^{\mu\nu}P_R b)F_{\mu\nu}, \\
\mathcal{O}_9 &= \frac{e^2}{(4\pi)^2}(\bar{s}\gamma^\mu P_L b)(\bar{\ell}\gamma_\mu \ell), \\
\mathcal{O}_{10} &= \frac{e^2}{(4\pi)^2}(\bar{s}\gamma^\mu P_L b)(\bar{\ell}\gamma_\mu\gamma_5 \ell),
\end{aligned} \tag{41}$$

---

[12] Other authors define mSUGRA as a smaller subspace of the CMSSM; see Ref. [175] for discussion and further references.

whereas others, such as

$$\mathcal{Q}_1 = \frac{e^2}{(4\pi)^2}(\bar{s}P_R b)(\bar{\ell}\,\ell),$$

$$\mathcal{Q}_2 = \frac{e^2}{(4\pi)^2}(\bar{s}P_R b)(\bar{\ell}\gamma_5\ell), \tag{42}$$

are almost exclusively the purvey of new physics. In general, the interesting quantities for new physics are therefore the differences between the expected SM and BSM values,

$$\Delta C_x \equiv C_{x,\text{BSM}} - C_{x,\text{SM}}. \tag{43}$$

More details can be found in the FlavBit paper [109] and Ref. [135].

WC: `Re_DeltaC7, Im_DeltaC7, Re_DeltaC9,`
`Im_DeltaC9, Re_DeltaC10, Im_DeltaC10,`
`Re_DeltaCQ1, Im_DeltaCQ1, Re_DeltaCQ2,`
`Im_DeltaCQ2.`

This model incorporates enhancements and suppressions to the real and imaginary parts of the Wilson coefficients of the effective operators $\mathcal{O}_7$, $\mathcal{O}_9$, $\mathcal{O}_{10}$, $\mathcal{Q}_1$ and $\mathcal{Q}_2$ (Eqs. 41, 42).

5.4.6 Nuisance parameters

These models contain values with significant uncertainties that can be essential for calculating signal rates (particularly in DM searches), but which are not part of a BSM model or the Standard Model.

Halo_gNFW: `alpha, beta, gamma, r_sun, rho0,`
`rhos, rs, v0, vesc, vrot.`
This as well as all other halo models specify the radial dark matter distribution $\rho(r)$ in the Milky Way and the local properties of dark matter relevant for direct detection and capture in the Sun. Specifically, this model corresponds to the generalized NFW profile

$$\rho(r) = \frac{2^{(\beta-\gamma)/\alpha}\rho_s}{(r/r_s)^{\gamma}\,[1+(r/r_s)^{\alpha}]^{(\beta-\gamma)/\alpha}}, \tag{44}$$

where γ (β) describes the inner (outer) slope of the profile, α is the shape in the transition region around the scale radius $r = r_s$, and $\rho_s \equiv \rho(r_s)$ is the scale density. Furthermore, the local properties of dark matter are described by means of the local density ρ_0 as well as a Maxwell-Boltzmann velocity distribution boosted to the rest frame of the Earth,

$$f(\mathbf{u}) = \frac{e^{-\left(\frac{\mathbf{u}+\mathbf{v}_{\text{LSR}}+v_{\odot,\text{pec}}+V_\oplus}{v_0}\right)^2}}{\pi^{3/2}v_0^3\,\text{erf}\left(\frac{v_{\text{esc}}}{v_0}\right) - 2\pi v_0^2 v_{\text{esc}}e^{-\left(\frac{v_{\text{esc}}}{v_0}\right)^2}}. \tag{45}$$

Here, v_0 is the most probable speed of a DM particle with respect to the galactic halo, while v_{esc} denotes the local escape velocity [176]. The remaining parameters describe the relative motion of the Earth and the Galactic rest frame: $\mathbf{v}_{\text{LSR}} = (0, v_{\text{rot}}, 0)$ is the motion of the Local Standard of Rest in Galactic coordinates, with v_{rot} being the local disk circular velocity, $\mathbf{v}_{\odot,\text{pec}} = (11, 12, 7)\,\text{km s}^{-1}$ is the peculiar velocity of the Sun [177], and $V_\oplus = 29.78\,\text{km}^{-1}$ is the Keplerian velocity of the Earth around the Sun. Notice that in this halo model the scale density ρ_s and the local density ρ_0 are treated as independent parameters.

Halo_gNFW_rho0: `alpha, beta, gamma, r_sun,`
`rho0, rs, v0, vesc, vrot.`
Same as **Halo_gNFW**, but deriving the scale density $\rho_s \equiv \rho(r_s)$ from a given value of the local density $\rho_0 \equiv \rho(r_{\text{sun}})$ via Eq. 44. Here, r_{sun} denotes the distance from the solar system to the Galactic center.

Halo_gNFW_rhos: `alpha, beta, gamma, r_sun,`
`rhos, rs, v0, vesc, vrot.`
Same as **Halo_gNFW**, but deriving the local density $\rho_0 \equiv \rho(r_{\text{sun}})$ from a given value of the scale density $\rho_s \equiv \rho(r_s)$ via Eq. 44.

Halo_Einasto: `alpha, r_sun, rho0, rhos, rs, v0,`
`vesc, vrot.`
Same as **Halo_gNFW**, but assuming instead the Einasto profile for the radial distribution of dark matter in the Milky Way:

$$\rho(r) = \rho_s \exp\left\{-\frac{2}{\alpha}\left[\left(\frac{r}{r_s}\right)^\alpha - 1\right]\right\}, \tag{46}$$

with r_s referring to the scale radius, ρ_s to the scale density, and α describing the shape of the profile.

Halo_Einasto_rho0: `alpha, r_sun, rho0, rs, v0,`
`vesc, vrot.`
Same as **Halo_gNFW_rho0**, but using the Einasto profile given by Eq. 46.

Halo_Einasto_rhos: `alpha, r_sun, rhos, rs, v0,`
`vesc, vrot.`
Same as **Halo_gNFW_rhos**, but using the Einasto profile given by Eq. 46.

nuclear_params_fnq: `deltad, deltas, deltau, fnd,`
`fns, fnu, fpd, fps, fpu.`
This model contains the nuclear matrix elements that parameterise the quark content of protons and neutrons, $f_{T_q}^{(N)}$, defined by

$$m_N f_{T_q}^{(N)} \equiv \langle N|m_q \bar{q}q|N \rangle, \tag{47}$$

where $N \in \{p, n\}$ and $q \in \{u, d, s\}$ [178]. The model also contains the parameters $\Delta_q^{(p)}$ that describe the spin content of the proton.

nuclear_params_sigma0_sigmal: `deltad`, `deltas`, `deltau`, `sigma0`, `sigmal`.
The same as **nuclear_params_fnq**, but with the 6 $f_{T_q}^{(N)}$ parameters replaced by the light quark content of the nucleon σ_l and the quantity σ_0, defined as

$$\sigma_l \equiv m_l \langle N|\bar{u}u + \bar{d}d|N \rangle, \tag{48}$$

$$\sigma_0 \equiv m_l \langle N|\bar{u}u + \bar{d}d - 2\bar{s}s|N \rangle, \tag{49}$$

where $m_l \equiv (1/2)(m_u + m_d)$. We take σ_l and σ_0 to be the same for protons and neutrons [179].

nuclear_params_sigmas_sigmal: `deltad`, `deltas`, `deltau`, `sigmal`, `sigmas`.
The same as **nuclear_params_fnq**, but with the 6 $f_{T_q}^{(N)}$ parameters replaced by σ_0 from Eq. 49 and the strange quark content of the nucleon σ_s, which is defined as

$$\sigma_s \equiv m_s \langle N|\bar{s}s|N \rangle. \tag{50}$$

Again, σ_0 and σ_s are assumed to be the same for protons and neutrons [179].

5.4.7 Toys

NormalDist: `mu`, `sigma`.
A simple test model consisting of two parameters: the width and central value of a Gaussian distribution. This model is used in most of the toy examples discussed in this paper.

TestModel1D: `x`.
A one-dimensional test model, typically used for debugging simple prior transformations, or when a dummy model is required (as in the external model example of the ColliderBit paper [108]).

demo_A, demo_B, etc:
These are additional example models available in the same header as NormalDist and TestModel1D, but commented out in order to keep from cluttering up the model hierarchy with fake models.

6 User interface and input file

In this section we describe the general user interface of GAMBIT. This includes a description of the available command line switches as well as a detailed walk-through of the structure and content of the main initialisation file. Further details

about the functionality of the dependency resolver, printers and scanners are given in the following sections.

6.1 Command line switches and general usage

GAMBIT is run by executing the `gambit` executable. The canonical way to launch a scan is to specify an initialisation file *myfile.yaml* with the `-f` switch, as in

```
gambit -f myfile.yaml
```

The full set of command-line switches available is:

`--version`
    Print the GAMBIT version number and exit.

`-h`/`--help`
    Display usage information and exit.

`-f` *file*
    Use instructions in *file* to start a scan.

`-v`/`--verbose`
    Run with full verbose output.

`-d`/`--dryrun`
    Perform a dry run of a scan. GAMBIT will resolve all dependencies and backend requirements, then list the function evaluation order to `stdout`, but won't actually start the scan. It will also produce necessary files and instructions for plotting the **dependency tree** (see Sect. 7). Requires `-f`.

`-r`/`--restart`
    Restart a scan, overwriting any existing output. Requires `-f`. If `-r` is not specified and previous output exists matching the instructions in *file*, GAMBIT will attempt to resume scanning based on that output.

GAMBIT also has various diagnostic modes that provide information about its current configuration from the command line. See Sect. 10.4 for further information.

6.2 The master initialisation file

The master initialisation file of GAMBIT is written in the YAML format.[13] YAML is a 'human-friendly, cross-language, Unicode-based data serialization language' that provides a general framework for setting up nested structures of common native data types. The format is reminiscent of

---

[13] See http://www.yaml.org for a definition of the standard. A compact introduction can be found at http://en.wikipedia.org/wiki/YAML. Note that GAMBIT is also fully compatible at the module level with the SLHA1 [180] and SLHA2 [167] formats for SUSY models; see Refs. [108–111] for details.

Python. As such, leading whitespace (i.e. the indentation level) matters, and is part of the syntax.

The top node of the master initialisation file is a dictionary that contains eight entries.

`Parameters` describes the scan parameters for different models.

`Priors` describes the priors to be placed on the scan parameters.

`ObsLikes` describes observables and likelihoods that the user would like to be calculated in a scan.

`Rules` specifies additional rules to guide the resolution of dependencies and backend requirements.

`Printer` provides details about how and where to store the results of the scan.

`Scanner` provides information about the scanning algorithm to be adopted in a scan.

`Logger` chooses options for logging GAMBIT messages during the scan.

`KeyValues` is an additional global option section.

Any number of other YAML files can be imported to any section of the master initialisation file, using the `!import` *other_file.yaml* directive. Imported files may import files of their own, and so on.

6.3 Model and parameters

6.3.1 General setup and fast priors

Selecting models to scan and setting options for their parameters is done in the `Parameters` section of the master YAML file, using the syntax:

```
Parameters:
  model_1:
    parameter_1:
        # optional fast prior statements
    parameter_2:
        # optional fast prior statements
    ...
  model_2:
    # content as above
  model_3:
  ...
```

For example, in the scalar singlet YAML file that ships with GAMBIT, `yaml_files/SingletDM.yaml`, this looks like:

```
Parameters:

 # SM non-Higgs parameters.
 StandardModel_SLHA2: !import
  include/StandardModel_SLHA2_scan.yaml

 # Nuclear matrix parameters.
 nuclear_params_sigmas_sigmal:
```

```
  sigmas:
    range: [19, 67]
  sigmal:
    range: [31, 85]
  deltau: 0.842
  deltad: -0.427
  deltas: -0.085

# SM Higgs-sector parameters
StandardModel_Higgs:
  mH:
    range: [124.1, 127.3]

# Scalar singlet dark matter parameters
SingletDM:
  mS:
    range: [45., 10000.]
    prior_type: log
  lambda_hS:
    range: [0.0001, 10.00]
    prior_type: log

# Dark matter halo parameters
Halo_gNFW_rho0:
  rho0:
    range: [0.2, 0.8]
  v0: 235.0
  vesc: 550.0
  vrot: 235.0
  rs: 20.0
  r_sun: 8.5
  alpha: 1
  beta: 3
  gamma: 1
```

Here we see that the SM parameters are imported from the YAML fragment `yaml_files/include/ StandardModel_ SLHA2_scan.yaml`.

As this layout suggests, multiple models can be scanned simultaneously; for example a particle physics model, plus a DM halo model, plus a set of nuclear physics parameters. This allows for arbitrary physics models to be combined easily and fluidly. This makes it simple to add new observables to existing scans even if they bring 'baggage' in the form of additional free parameters. The typical example is that of nuisance parameters. Adding an observable that depends not only on the particle physics scenario, but also the assumed value of the top mass, for example, is easy: one adds the new observable to the `ObsLikes` section, and adds the value or range of top masses to consider when calculating that observable to the `Parameters` section. Broader examples of the utility of this arrangement include observables with dual implications for both particle physics and cosmology, or for both BSM and neutrino physics.

The subsection following each parameter is an optional 'fast prior' definition. For the purposes of sampling parameter values, a prior is the portion of the probability distribution function for choosing parameter values that is independent of the likelihood, i.e. the sampling distribution determined *prior* to any contact with data. Many sampling algorithms

(indeed, essentially all useful ones) apply *additional* conditions designed to preferentially sample points that constitute better fits to data – but one must always choose what initial prior to employ, independent of the sampling algorithm to be employed. The simplest example would be assigning independent flat distributions for each parameter, viz. 'flat priors'. When paired with a naive random scanner, this would lead to simple uniform sampling of the parameter values.

Using the `Prior` section (see Sect. 6.3.2 and Ref. [112]), GAMBIT makes it possible to use any arbitrary prior in a scan – but in most cases a very simple prior will suffice. The fast prior subsection provides a streamlined way to directly set such simple priors in the `Parameters` section, for each parameter.

Note that every parameter of every model mentioned in the `Parameters` section must be associated with some prior, either in the `Priors` section or via a fast prior in the `Parameters` section. This applies even if the parameter is not actually used in a given scan. In this case, the parameter should normally simply be set to some arbitrary constant value in the YAML file.

In it's simplest form, the fast prior section can just specify such a value to fix a parameter to during a scan (a so-called 'delta-function prior'):

```
model_1:
    parameter_1: 125.0
    parameter_2: 750.0
```

The same thing can be achieved with

```
model_1:
    parameter_1:
        fixed_value: 125.0
    parameter_2:
        fixed_value: 750.0
```

This syntax naturally extends to specifying an ordered set of points to cycle through, e.g.

```
model_1:
    parameter_1: [125.0, 142.5, 119.0]
    parameter_2:
        fixed_value: [750.0, 2015.0, 38.0]
```

There may be cases where parameters spanning multiple models are equivalent and should thus be described as a single parameter. GAMBIT allows model parameters to be combined using the `same_as` keyword. Thus, *parameter_1* of *model_1* can be set to be equal to *parameter_2* of *model_2* via a fast prior entry such as

```
model_1:
    parameter_1:
        same_as: model_2::parameter_2
        scale: scale
        shift: shift
```

Here, *model_1::parameter_1* will be automatically set from the value assigned to *model_2::parameter_2* at each point in the scan. The keywords `scale` and `shift` can also be optionally specified; these scale the parameter by an amount *scale* and shift it by *shift*. Thus, in the above example,

$$model\_1 :: parameter\_1 =$$
$$shift + scale * model\_2 :: parameter\_2. \tag{51}$$

When two models being scanned have parameter names in common, extra care needs to be taken. ScannerBit treats each parameter of each model as fully separate by default, but any module functions that declare both models as allowed (either individually or in combination) will trigger a runtime error when GAMBIT attempts to add the values of all parameters in both models to the `Params` pipe (cf. Sect. 3.2.3). Usually this indicates poor module function design, although there are some use cases, where the `same_as` directive is in use, when it may be simplest to proceed without worrying which of the two models' common parameters appears in the `Params` pipe. Users wishing to hack their way through such a situation can set the `ALLOW_DUPLICATES_IN_PARAMS_MAP` precompiler variable in `Elements/include/gambit/Elements/module_macros_incore.hpp` to `1`.

Other fast priors can be chosen via the `prior_type` keyword, which can be set to `flat`, `log` (uniform in the log of the parameter value), or various trigonometric functions (`cos`, `sin`, `tan` or `cot`), as in

```
model_1:
    parameter_1:
        prior_type: chosen_prior
        range: [low, high]
    parameter_2:
        prior_type: log
        range: [5, 75]
```

The allowed values of the parameters are given by setting `range`. The `scale` and `shift` parameters also work with `prior_type`, in just that same way as with `same_as`.

If no fixed value is given for a parameter, and both `prior_type` and `same_as` are absent but `range` is given, a `flat` prior is assumed.

Additional custom priors can be be written as plugins for ScannerBit, and accessed by setting `prior_type:plugin`; details can be found in Ref. [112].

6.3.2 More involved priors

Certain priors introduce correlations between parameters. This makes specifying a separate, unique prior for each parameter impossible. Such multidimensional priors, operating on multiple parameters simultaneously, can only be declared in a separate `Prior` section of the main YAML file.

```
Priors:
  prior_name:
    parameters: [model_1::param1, model_1::param2,
                 ...]
    prior_type: prior_type_1
    options
  other_prior_name:
    parameters: [model_2::paramA, model_2::paramB,
                 ...]
    prior_type: prior_type_2
    options
  ...
```

A multidimensional prior is defined under a new user-defined key such as *prior_name*. Each prior declared in this way must specify a vector of input parameters, a prior type, and any options required by the prior. A list of prior types and their options can be obtained with the **GAMBIT** diagnostic `gambit` priors (see Sect. 10.4.7). Available multidimensional priors include Gaussian and Cauchy distributions, as well as the ability to specify any additional **ScannerBit** prior plugin present on a user's system; these are discussed in detail in Ref. [112].

6.4 `ObsLikes`: target observables and likelihoods

Entries in this section determine what is calculated during a scan. Each entry lists a likelihood contribution or an observable that should be calculated during the scan. (Likelihood functions and observables are largely the same within **GAMBIT**, the main difference being that the former are used to drive the scan, whereas the latter are simply recorded.) The minimal allowed entry has the form

```
ObsLikes:
  - capability: example_capability
    purpose: example_purpose
  - ...
```

Here, *example_capability* is the capability of the likelihood or observable to be calculated, while *example_purpose* is its role in the scan. The latter determines its treatment by the scanner and the printer system. In the simplest cases, `purpose` will be set to either `LogLike` or `Observable`.[14] In the case of a `LogLike`, the calculated quantity will be used as one of the likelihoods in the scan. As a convention in **GAMBIT**, all likelihoods are given in terms of $\log \mathcal{L} = \ln(\text{likelihood})$. In the case of an `Observable`, the calculated quantity will be simply written as additional output and will be available for later post-processing.

For example, the following entries from `yaml_files/SingletDM.yaml` ensure that the likelihood from the dark matter relic density is included in the overall likelihood function, and that the value of the relic density itself is saved in

---

[14] Alternative purposes are relatively easy to arrange, but these are the conventional ones. See Sect. 8 for further discussion.

the output of the scan, for every valid combination of model parameters:

```
ObsLikes:

  # Relic density likelihood contribution
  - capability: lnL_oh2
    purpose: LogLike

  # Relic density prediction
  - capability: RD_oh2
    purpose: Observable
```

It will often happen that several module functions can provide the same capability. In order to remove such ambiguities, it is possible to specify the requested quantity further by adding one or more of the following optional arguments

```
ObsLikes:
  - capability: capability
    purpose: purpose
    type: type
    function: function
    module: module
  - ...
```

Here, *type* specifies the **C++** type of the module function that should be used to fulfil the requested capability, *function* explicitly gives the name of a module function, and *module* demands that the function must come from a specific module. These additional specifications in the `ObsLikes` section are in fact just a convenient shortcut for setting up the most common **rules** for dependency resolution. Dependency resolution rules can be set up in far more generality in the separate `Rules` section, which we discuss below.

In the case of the purpose `LogLike`, the *type* of the module function selected *must* be `double`, `float`, `std::vector<double>` or `std::vector<float>`, as the result will be sent to the **likelihood container** to contribute to the total likelihood function. (This applies regardless of whether the user has specified the *type* explicitly, or left it to the dependency resolver to work out.) In the case of vectors, the likelihood container automatically sums all entries.

Finally, the additional option `printme` can be set for each `ObsLikes` entry, for example

```
ObsLikes:
  - capability: example_capability
    purpose: example_purpose
    printme: true
  - ...
```

This option is `true` by default, meaning that by default **GAMBIT** will attempt to record to disk (i.e. 'print'; see Sect. 9) the computed result of the each of the target observables/likelihoods. This is the behaviour that one almost always wants during a production scan, however by setting `printme` to `false` the user can tell **GAMBIT** not to try to output the result of the thusly-flagged computation. It is useful to do

this when testing and debugging new module functions, for example, because these often produce results that are not of a printable C++ type (and so attempting to print them would cause a runtime error, see Sect. 9.3), yet one will often want to set these functions as `ObsLikes` targets just to ensure that GAMBIT will run them.

6.5 Rules: dependency resolution and module options

Entries in the `Rules` section determine the details of how the likelihoods and observables listed in the `ObsLikes` section are calculated in the scan. In the rather common case that several different module functions can provide a capability requested in the `ObsLikes` section, or several module functions can provide the neccessary capability-type pair requested in another module function's **dependency**, then further specifications in the `Rules` section are required to fully define the scan. The `Rules` section can likewise be used to control the resolution of backend requirements, and to set options for individual module functions, modules and backend initialisation functions.

6.5.1 Module function dependencies

In the rather common case that several different module functions provide the same requested quantity, further rules are necessary to define the scan. Note that with quantity, we refer here specifically to capability/type pairs, $quantity \equiv (capability, type)$. These rules can be specified in the `Rules` section of the initialisation file. Furthermore, this section is used to control the resolution of backend dependencies, and to set options for individual module functions, modules and backend initialisation functions. In this sense, the rules determine how an individual point is calculated during the scan.

In the simplest case, a rule has the form

```
Rules:
  - capability: capability
    type: type
    function: function
    module: module
```

where `capability` is required, `type` is optional, and one or both of the entries `function` and `module` must be given. This entry translates into the rule: Any *capability* with C++ type *type* should be resolved by module function *function* from the module *module*. Assigning the empty string `""` or the wildcard character `"*"` to an entry is equivalent to omitting it. If regex is activated (this is *not* the default; see Sect. 6.9), all entries are actually treated as regular expressions, allowing rules to be made arbitrarily complex.[15]

---

[15] For details about regular expressions we refer the reader to https://en.wikipedia.org/wiki/Regular_expression.

A simple example of such a rule is the one in `yaml_files/SingletDM.yaml` that specifies that the observed relic density should be treated as an upper limit only when computing the likelihood. This allows for the possibility that some of the dark matter is not in the form of scalar singlet particles.

```
# Choose to implement the relic density likelihood
# as an upper bound, not a detection
- capability: lnL_oh2
  function: lnL_oh2_upperlimit
```

This rule says that wherever the capability `lnL_oh2` is needed in a scan, GAMBIT must use a function with the name `lnL_oh2_upperlimit`. As it turns out, there is only one function with such a name in GAMBIT 1.0.0, and it lives in DarkBit – so this rule forces `DarkBit::lnL_oh2_upperlimit` to be used.

The simple form shown above applies a rule to the resolution of dependencies of *any* module functions matching the specified `capability` and `type`. In order to set up rules that only affect the dependency resolution of a specific module function, one can add a dedicated `dependencies` subsection, and optionally omit any of the top-level keys `capability`, `type`, `function` and `module` (or equivalently, set them to `""` or `"*"`).

```
Rules:
  - capability: capability
    type: type
    function: function
    module: module
    dependencies:
    - {capability: cap_A, type: type_A,
       function: func_A, module: mod_A}
    - {capability: cap_B, type: type_B,
       function: func_B, module: mod_B}
    - ...
  - ...
```

If regex is activated, the values are treated as regular expressions. The entry translates into the following rule: when resolving dependencies of module function *function* in module *module*, which provides capability *capability* with C++ type *type*, apply the rules listed under the keyword `dependencies`.

If conflicting rules are found during dependency resolution, GAMBIT will throw an error. This is intended to reduce side effects that changes in some parts of the initialisation file can have on other parts. However, rules can be *explicitly* declared as weak and over-rideable, by using the flag weak!, as per

```
Rules:
  - !weak
    capability: capability
    type: type
    function: function
    module: module
  - ...
```

Note that the flag affects the *entire* rule for which it is set, not only specific nearby keywords.

A special case can occur if several module functions depend on the same quantity as they provide. In this case these module functions can be chained, and setting up such chains in the rules section is simplified by using the keyword `functionChain`. This is illustrated in the following example, where *func1*, *func2* and *func3* are supposed to provide as well as depend on *capability* with *type*. These functions will be chained together, with *func1* fulfulling the depenencies of *func2* etc.

```
Rules:
 - capability: capability
   type: type
   functionChain: [func1, func2, func3]
   module: module
```

Finally, when performing `type` matching, the dependency resolver takes all type equivalences defined in `config/resolution_type_equivalency_classes.yaml` into account. We discuss this type equivalency database for dependency resolution in more detail in Sect. 10.5.

6.5.2 Backend requirements

After a module function has been selected to take part in a scan, its backend requirements are resolved. This process can be guided and controlled using rules for backend requirements, which have the form:

```
Rules:
 - capability: capability
   type: type
   function: function
   module: module
   backends:
   - {capability: cap_A, type: type_A,
      function: func_A, backend: backend_A,
      version: backend_A_version_number}
   - {capability: cap_B, type: type_B,
      function: func_B, backend: backend_B,
      version: backend_B_version_number}
   - ...
```

The usage is essentially identical to the one discussed above for dependencies, except that `backend` may be specified rather than `module`, and a specific version of a backend may be requested, as e.g.

```
 - capability: Higgs_Couplings
   backends:
   - {backend: FeynHiggs, version: 2.11.3}
```

There are also a number of other restrictions that can be applied via rules declared in the module function's rollcall header entry (Sect. 3.1.3). These include backend requirements that are only activated for specific models (analogous to model-conditional dependencies of module functions), restricted lists of permitted backends and versions, and the condition that certain combinations of backend requirements must be resolved by the *same* version of the *same* backend.

6.5.3 Options for module functions

Besides setting rules for the resolution of dependencies and backend requirements, the `Rules` section can also be used to set options for module functions. This is done with the `options` keyword, as

```
Rules:
 - capability: capability
   type: type
   function: function
   module: module
   options:
     key_A: value_A
     key_B: value_B
     ...
 - ...
```

This rule sets the option *key_A* to *value_A* and option *key_B* to *value_B*, for any module function that matches the indicated *capability*, *type*, *function* and *module*. Any of these keywords can be omitted; if regex is activated, they are treated as regular expressions. This allows, for instance, module-wide options to be set using just the name of the module, whilst omitting the other three keywords or setting them to wildcards:

```
Rules:
 - module: DarkBit
   options:
     DM_is_made_of: axions
```

Here, the key `DM_is_made_of` is accessible by all module functions in the module `DarkBit`.

This last example is a bit glib, as in reality `DM_is_made_of` is not a recognised option of any functions in **DarkBit**, so setting it doesn't actually have any effect in **GAMBIT** 1.0.0. A more realistic example is:

```
Rules:
 # Use the DarkBit native calculator
 # to compute the relic density
 - capability: RD_oh2
   function: RD_oh2_general
   options:
     fast: 1
```

This can be seen in e.g. `yaml_files/SingletDM.yaml`. This rule specifically selects the `RD_oh2_general` function from **DarkBit** for calculating capability `RD_oh2` (i.e. the relic density), and passes it the option `fast = 1`, to set the accuracy required when solving the Boltzmann Equation for the thermal relic density of scalar singlet particles.

The key-value pairs specified in this way are easily accessed by any module function that matches a given rule, using `runOptions->getValue` (cf. Sect. 3.2.4).

In most cases, module functions will interpret option values as simple C++ types (commonly `float`, `int`, `bool` or `std::string`), but composite types like `std::vector<double>` can also be set. The necessary syntax for doing this is defined by the YAML standard. Options can also be easily nested, with the C++ type of the top-level option to be retrieved itself a `YAML::Node`,[16] from which lower-level options can then be retrieved.

Information about what options are available for which module function can be found in the module function documentation. Options that are never requested by module functions at runtime are silently ignored.

In case of ambiguity, such as when an option requested by a module function is listed in several matching rules, GAMBIT throws an error during initialisation.

6.6 Printer

The GAMBIT "printer" system handles the output of all scan results, whether to disk, a network resource or any other output stream. This system allows all GAMBIT output to be handled in an abstract way throughout the code, with the actual format of the output being decided by the choice of an output plugin (a **printer**) at runtime, via the master YAML file. Therefore, setting up GAMBIT output consists primarily of choosing a printer and setting options for it. In this section we describe how to do this; full details of the printer system can be found in Sect. 9.

Note that output handled by the GAMBIT printer system is essentially independent of other output that might be created by any backend or scanner codes. This allows the output to remain as uniform as possible, regardless of the scanning algorithm and external codes being used.

GAMBIT 1.0.0 ships with two printers: ascii and hdf5. The ascii printer outputs data as a simple ASCII table, whereas the hdf5 printer writes data to a binary file in HDF5 format.[17] The former format is useful for its simplicity, however the latter is far superior when dealing with large datasets, particularly in terms of disk usage and read/write speed. We have also upgraded the external analysis tool pippi [181] to accept GAMBIT input in these formats; it can be easily retrieved via the GAMBIT build system (Sect. 11.3).

Most options that affect the output system are entered in the `Printer` section of the master YAML file. The basic layout of this section is:

```
Printer:
  printer: plugin_name
  options:
    option_1: value_1
    option_2: value_2
    ...
```

That is, one chooses a plugin *plugin_name* and sets its options, which vary with the plugin. In the next sections we describe the options available in each printer.

6.6.1 Common options

These options are common to both the ascii and hdf5 printers:

```
options:
  output_path: default_output_path
  output_file: filename
```

`output_path` specifies the directory in which the printer output will be stored. By default it is set to the value of `default_output_path` as set in the `KeyValues` section of the input YAML file (see Sect. 6.9), however if a value is set here it will override that default.

`output_file` specifies the name of the file in which to store data generated during the run. If it does not exist then it will be created.

6.6.2 Specific options: ascii printer

The only specific option for this plugin is `buffer_length`, which defaults to a value of `100`:

```
Printer:
  printer: ascii
  options:
    buffer_length: 100
```

This specifies the size of the internal buffer used by the printer. A value of N will cause output to be written to disk every N model points. If model points are slow to evaluate, it can be useful (particularly during testing) to set `buffer_length` to `1` so that output is generated frequently. However, if model points are evaluated extremely rapidly then frequent writing of output will create a significant bottleneck, and a high value of `buffer_length` will be more appropriate.

---

[16] This class is defined in the contributed package yaml-cpp, which ships with GAMBIT. Documentation is available at http://github.com/jbeder/yaml-cpp.

[17] https://www.hdfgroup.org/HDF5/.

6.6.3 Specific options: *hdf5* printer

There are three specific options for this plugin:

```
Printer:
  printer: hdf5
  options:
    group: "/"
    delete_file_on_restart: false
```

The first is `group`, which defaults to `"/"`. This option specifies the name of the group *within* the host HDF5 `output_file` in which data will be stored. HDF5 files are structured similarly to a filesystem (i.e. hierarchically) and a 'group' is analogous to a directory. Various objects (such as datasets, and other groups) are then stored within groups[18] The default value of `"/"` specifies the root group, and this option should rarely need to be set to anything else. A deeper-layer group can be specified e.g. as `"/group1/group2/etc/"`. Absent groups at any layer will be automatically created.

The second option is `delete_file_on_restart`. This option is mainly a convenience for performing repeated test scans, and causes the file specified by `output_file` to be deleted if it already exists when a run restarts (i.e. if the `-r` command line flag is used, see Sect. 6.1). By default this is false, meaning that if a HDF5 file already exists matching the name given in `output_file` then GAMBIT will attempt to *add* the data for the run to this pre-existing file.

Further details of the HDF5 objects that GAMBIT writes to disk via this printer can be found in Sect. 9. Note that results from several runs can be stored inside the same HDF5 file by storing the data in different groups, however it is safer to use separate files because HDF5 files are vulnerable to corruption from write errors (which in principle can occur if GAMBIT terminates abnormally; see Sect. 10.8 for safe early shutdown methods), and data recovery is difficult. If `delete_file_on_restart` is **false** and the chosen `group` already exists, GAMBIT will throw a runtime error telling you to choose a different group or overwrite the whole file. Groups can be deleted, however the disk space they occupy cannot be reclaimed without copying the entire contents of the HDF5 file into a new file, e.g. using the **h5repack** command line tool.[19] We leave these kind of file manipulations to the user.

6.6.4 Output selection

The outputs handled by the printer system are simply the results of module function evaluations. However, not all module function results are of a C++ type that can be 'printed'

---

[18] See https://www.hdfgroup.org/HDF5/doc/Glossary.html for further description of 'groups' and 'datasets' in HDF5.

[19] https://www.hdfgroup.org/HDF5/doc/RM/Tools.html#Tools-Repack.

with every printer (see Sect. 9.3 for the restrictions), so GAMBIT cannot automatically output *all* results. To instruct GAMBIT to write the result of a calculation to an output stream, the module function that computes a result must be selected to fulfil one of the capabilities requested in the `ObsLikes` section of the master YAML file. Intermediate results, computed by functions run by the dependency resolver only in order to fulfil dependencies of other functions, are not output.

6.7 Scanner

GAMBIT ships with a variety of **scanner plugins** that can be used in a "plug and play" manner. A full list of scanner plugins can be obtained from the GAMBIT scanners diagnostic (Sect. 10.4.5). A scanner is selected by specifying one of these plugins and any plugin-specific options in the `Scanner` section of the YAML file, e.g.

```
Scanner:
  use_scanner: nested_sampler
  scanners:
    nested_sampler:
      plugin: MultiNest
      like: LogLike
      nlive: 4000
      tol: 0.5
      mmodal: 1
    other_sampler:
      plugin: ...
      ...
    ...
```

The `Scanner` section can contain multiple scanner definitions with user-defined names, such as `nested_sampler` and `other_sampler` in the above example. The scanner that will actually be used in a given scan is specified with the `use_scanner` key. Within the YAML scanner definitions, the `plugin` option must be set to a valid scanner plugin known to GAMBIT, and any necessary/desired options for that scanner should also be set. Note that a typical scanner plugin requires a **purpose** to use for its objective function, such as `LogLike` or `Observable`; this is provided by setting the `like` option in the example of the `MultiNest` plugin. Valid and required plugin options, plugin descriptions, and the status of a plugin can be obtained through the GAMBIT free-form diagnostic (see Sect. 10.4.8),

```
gambit plugin_name
```

where *plugin_name* is the name of the scanner plugin.

GAMBIT also ships with a number of simple objective test functions, which can be used as objective functions for a scan in place of the regular GAMBIT **likelihood container** output, for testing scanners and other parts of the code. These exist as **test function plugins** in ScannerBit, and are accessed from the main YAML file with similar syntax to scanners, e.g.

```
Scanner:
  use_objectives: my_test_function
  objectives:
    my_test_function:
      plugin: uniform
      parameter_A: 10
      parameter_B: false
    other_test_function:
      plugin: ...
      ...
    ...
```

As the `use_objectives` directive suggests, multiple test functions can be specified with the regular YAML $[x,y]$ syntax if desired, in which case all the listed objectives will be multiplied to form the actual objective function to be used in the scan. Details of the available test functions and their options can be found in the ScannerBit paper [112].

6.8 Logger

The logging output of a scan can be directed to various output files. This is done using entries of the form:

```
Logger:
  prefix: output_path
  redirection:
    [Scanner, Warning] : "scanner_warnings.log"
    [ExampleBit_A] : "ExampleBit_A.log"
    ...
```

Here `prefix` specifies the output location for log files (defaulting to `default_output_path`; cf. Sect. 6.9), and the entries in the `redirection` subsection dictate which logging messages go to which file. These options are discussed further in Sect. 10.2.

6.9 `KeyValues`: general purpose options

Most of the general behaviour of GAMBIT is controlled by various options in the `KeyValues` section. The syntax is the same as described above in the context of the module function options. We provide here a complete list of available options. Where we indicate concrete values, these are the default values that will be used if the option is omitted; where no default is indicated, the option is required.

```
KeyValues:

  likelihood:
    # The value of the log-likelihood to assign to
    # invalid points. Also the log-likelihood value
    # below which an otherwise valid point is declared
    # invalid.
    model_invalid_for_lnlike_below: lnlike_min
    # Alternative value of the log-likelihood to
    # assign to invalid points later in a scan (e.g.
    # with the MultiNest scanner; see  [112]).
    model_invalid_for_lnlike_below_alt: #defaults to
    # 0.5*lnlike_min.

    # Print likelihood debug information to stdout and
    # logs, including parameter values and
    # contributions of individual likelihood
    # components. Set true automatically if the master
    # debug flag (below) is true.
    debug: false

  exceptions:
    # Set the fatality of different exceptions (see
    # Sec. 10.3). By default, all
    # errors are fatal and all warnings non-fatal.
    core_warning: non-fatal
    core_error: fatal
    ExampleBit_A_warning: non-fatal
    ExampleBit_A_error: non-fatal
    ...

  dependency_resolution:
    # If multiple module functions can resolve the
    # same dependency, prefer the one that is more
    # tailored for the scanned model. See Sec. 7.1.
    prefer_model_specific_functions: true
    # Interpret rules in terms of regular expressions
    use_regex: false
    # Print running average runtime for all functions
    # in dependency resolver logs
    log_runtime: false

  # Print timing information into hdf5 output
  print_timing_data: false

  # Root prefix to use in all output paths. The
  # default value is based on the input \YAML file
  # name, with the (final) file extension removed.
  default_output_path: "runs/inifile_name/"

  # Call MPI_ABORT when attempting to shut down. Many
  # implementations of MPI_ABORT are buggy and do not
  # abort other MPI processes properly; in these
  # cases, set this option false to let GAMBIT try to
  # abort things its own way.
  use_mpi_abort: true

  # Pick a random number generator engine.
  # See Sec. 10.6 for details.
  rng: default # default = mt19937_64 in GAMBIT 1.0.0

  # Turn on master debug mode. Implies
  # Logger:debug=true and
  # KeyValues:likelihood:debug=true
  debug: false
```

7 Dependency resolver

The **dependency resolver** runs during the initialisation stage of a GAMBIT scan. It determines which module functions are required for a specific scan, infers their initial evaluation order, and connects their **pipes**. A major part of this plumbing exercise is constructing the **dependency tree** of a scan, a directed acyclic graph with dependency pipes as the connectors ('edges' in graph language) and module functions as the nodes. Roughly speaking, the dependency tree starts at its 'top' with the scanned models and their parameters, and terminates at the 'bottom' with functions that provide the likelihoods and observables requested in the `ObsLikes` section of the scan's initialisation file (Sect. 6.4). An example can be seen in Fig. 5. The construction of a valid dependency tree will happen mostly automatically, and depends only on the declarations in the module and backend **rollcall headers**. However, it is rather common in GAMBIT that there are several ways to calculate the same thing, in which case additional rules have to be specified in the input file (Sect. 6.5).

7.1 General procedure

The steps of dependency resolution are:

1. Disable all module and backend functions not compatible with the models being scanned.
2. Based on the entries of the `ObsLikes` section, make a list of initially requested quantities; this is the initial dependency queue.
3. Pick an unresolved quantity from the dependency queue, along with a designated target. Entries in the initial dependency queue can be thought of as having the chosen printer as their target.
4. Make a list of module functions that can provide the requested quantity.
5. If the `KeyValues` entry `prefer_model_specific_functions` is **true**:

   – If any module functions on the list are tailor-made for the scanned models, remove all other module functions from the list.
   – If any module functions on the list are tailor-made for ancestors of the scanned models, keep only the module functions most closely related to the scanned models.

6. Adopt the `Rules` specified in the initialisation file (see Sect. 6.5), removing non-matching module functions from the list.
7. If exactly one module function is left on the list, resolve the quantity requested by the target function with the capability provided by that module function. This auto-

matically connects the pipe of the target function to the result of the resolving function.

8. If the resolving function was not already activated for the scan, activate it and add its dependencies to the dependency queue (with the resolving function as new target function).
9. Resolve backend requirements, as described below.
10. Resolve module function options, as described below.
11. Repeat from step 3 until the dependency queue is empty.

7.2 Evaluation order

After building up the dependency tree of module functions, the dependency resolver determines the initial runtime ordering of its chosen module functions. An obvious minimal requirement is that if the output of module function $\mathcal{A}$ is required by module function $\mathcal{B}$, then $\mathcal{A}$ must be evaluated before $\mathcal{B}$. We do this by topologically sorting the directed dependency tree, using graph-theoretic methods from the Boost Graph Library.[20]

In most cases, the evaluation order of the observables and likelihoods listed in the `ObsLikes` section (Sect. 6.4) remains unconstrained by the topological sorting. The dependency resolver first orders the likelihoods by estimating the expected evaluation time for each one, including all depen-

---

[20] http://www.boost.org/doc/libs/1_63_0/libs/graph/doc/. The reader may wonder how mutually-dependent quantities should be dealt with, i.e. in cases where the values of $\mathcal{A}$ and $\mathcal{B}$ are defined by a set of equations that must be solved simultaneously, by means of iteration. Take the calculation of precision values of m_W and m_h in the MSSM for example, where each depends on the other. GAMBIT does not provide any option for doing such iterative calculations directly through the dependency tree. Generally the way to deal with such a situation is to either

1. write a module function that can compute the two quantities simultaneously and self-consistently (i.e. that does the iteration internally), returning them both as its result, or
2. use function pointers as return values of module functions.

For option 2, take the Higgs mass example. If a module author wishes to permit the user to choose from two possible expressions for m_h that depend on m_W, they would first write the two expressions as functions of m_W. Call these expressions $f(m_W)$ and $g(m_W)$. The author would then write one or more module functions that return a pointer to f or g. The module function that computes m_W should then depend on a pointer to a Higgs mass function, and then just call it (either f or g, depending on which one the user chooses) whilst it does its iterative calculation of m_W. It should then return its final value of m_W. Another module function responsible for computing m_h should then depend on both the value of m_W, and the pointer to the same Higgs mass function (f or g). This module function then simply takes the previously computed value of m_W, passes it to the function pointed to by its dependency on the Higgs mass function pointer, retrieves the final value of the Higgs mass, and returns it as its own result.
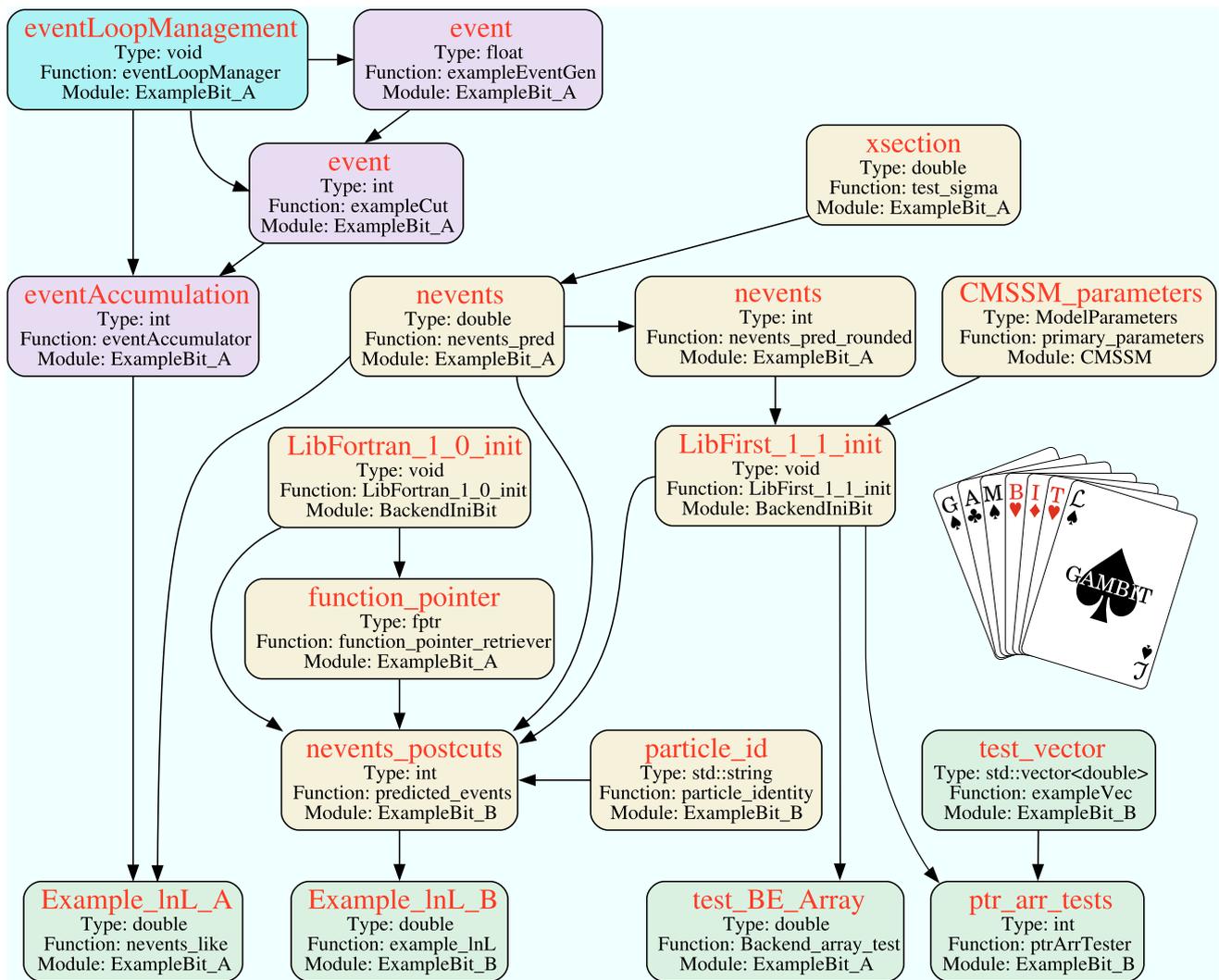
Fig. 5 An example **dependency tree** generated in the initialisation stage of a GAMBIT scan. Each block corresponds to a single **module function**, with the red text indicating its **capability**. Arrows indicate resolution of **dependencies** of different module functions with the results of others. The functions selected by the **dependency resolver** to provide the observables and likelihoods requested in the `ObsLikes` section of the scan's input YAML file are shaded in green. Module functions shown shaded in purple are **nested module functions**. These run in an automatically-parallelised loop managed by a **loop manager** function, which is shown shaded in blue. This example is included in the GAMBIT distribution as `spartan.yaml`; see Sect. 12.1 for more details. Figures like this can be generated for any scan by following the instructions provided after calling GAMBIT with the `-d` switch; see Sect. 6.1 for details

dent module functions, along with the probability that each likelihood will invalidate a point. (A point may be invalidated if the likelihood is extremely close to zero, the point is unphysical, etc.) These estimates are based on the runtime and invalidation frequency of the previously calculated points, and updated on the fly during the scan. The dependency resolver then sorts the evaluation order of likelihoods such that the expected average time until a point is invalidated is minimised. In practice this means that, for instance, the relatively fast checks for consistency of a model with physicality constraints, such as perturbativity and the absence of tachyons, would be automatically performed before the often time-consuming evaluation of collider constraints. This gives a significant efficiency gain in a large scan, because expensive likelihoods are not even evaluated for points found to be invalid or sufficiently unlikely on the basis of faster likelihoods.

Observables not associated with likelihoods used to drive a scan (cf. Sect. 6.4) are always calculated after the likelihood components, as they do not have the power to completely invalidate a model point. Invalid observable calculations can still be flagged, but they will not trigger the termination of all remaining calculations for that point in the way that an invalid likelihood component will.

7.3 Resolution of backend requirements

Resolving backend requirements is in some sense a lot easier than resolving module function dependencies, in that backend requirements cannot themselves have explicit backend requirements nor dependencies, so there is no equivalent of the dependency tree to build. However, the ability to specify groups of backend functions from which only one requirement must be resolved, along with rules that apply to them (Sect. 3.1.3), especially the declaration that backend requirements that share a certain tag must be resolved from the same backend – without necessarily specifying *which* backend – makes backend resolution a uniquely challenging problem.

The dependency resolver employs an iterative approach to backend resolution. It performs multiple passes over the list of backend requirements, choosing to defer resolution of ambiguous requirements until resolution of other requirements makes it possible to uniquely resolve the initial requirements. The overall strategy proceeds as follows:

1. Create a new group of backend requirements, consisting of all requirements that were declared in the rollcall header without a group. This will be a special group; unlike declared groups, all requirements in this group must be resolved rather just one.
2. Create a queue of all groups of backend requirements.
3. Choose a group from the queue.
4. (a) If the group is a regular group, iterate through all available backend functions and retain those that fulfil all rules of the group. If no backend function exists that satisfies all rules, throw a runtime error. If only one such function exists, resolve the group backend requirement with it. If multiple solutions are found, but one or more of them is subject to a rule linking it to another backend requirement, flag the group as one whose resolution should be deferred until other backends have been resolved.
   (b) If instead the group is the special one, iterate over all requirements in the group, attempting one by one to find a unique backend function that fulfils each requirement. Fail if no solution exists to any one of these requirements. If just one solution exists to a given requirement, resolve the requirement with it. If no unique solution is found for some requirement, but one or more candidates is subject to a rule linking it to another requirement, flag the group for deferral and come back to its unresolved members later.
5. If it has been flagged for deferral, add the group again to the end of the queue.
6. Repeat from step 3 until either

   (a) all groups have been fully resolved, or

   (b) the queue stagnates, i.e. a full iteration has been carried out through the entire queue of groups without any successful backend resolutions. In this case, disable the possibility to defer resolution, and try one last iteration through the queue, ultimately failing if any backend groups fail to resolve on the final attempt.

7.4 Resolution of loops and nested functions

As discussed in Sect. 3.1.4, it is possible to write special module functions (**loop managers**) that control the parallel execution of other module functions (**nested module functions**). Nested functions explicitly declare a dependency on a loop manager with a certain capability. The dependency resolution proceeds then as for non-nested module functions. The main difference is that loop managers have access to pointers to the nested module functions that they control. The dependency resolver generates a miniature dependency tree for each loop manager, consisting of all nested functions assigned to run inside the loop managed by that manager. The loop manager is then given responsibility for executing the nested functions, in the order provided to it by the dependency resolver. Further details can be found in Sect. 3.1.4.

7.5 Option resolution

Each time a module function is activated during the process of dependency resolution, the dependency resolver searches the `Rules` section of the initialisation file for relevant option entries (see Sect. 6.5.3 for the format of option entries). All options matching the characteristics of the activated module function are collected into a new object, which is then connected to the function's `runOptions` pipe (cf. Sect. 3.2.4). If the same option is set to conflicting values in multiple entries in the `Rules` section of the initialisation file, the dependency resolver will throw an error.

8 Statistics and scanning

In this section we explain the statistical strategy employed by GAMBIT (Sect. 8.1), how to obtain final inferences from its outputs (Sect. 8.2), and the generic likelihood forms available within GAMBIT for use by module functions that do not define their own dedicated likelihoods (Sect. 8.3).

8.1 The role of ScannerBit

To launch a GAMBIT run, a user requests a parameter scan of a certain model, specifying ranges and priors of the model

parameters, how to sample them, and the quantities that should be calculated and included in the scan. The GAMBIT model database activates the relevant model ancestry, which the dependency resolver uses together with the capabilities and types of the user's requested quantities to select and connect appropriate module and backend functions into a dependency tree. Choosing which values of the model parameters to run through this dependency tree is the job of ScannerBit, the sampling and statistics module [112]. ScannerBit applies any prior transformations requested by the user, and activates the appropriate **scanner plugin** in order to run the requested sampling algorithm. ScannerBit presently contains plugins for nested sampling (MultiNest [182]), Markov Chain Monte Carlo (GreAT [183]), a population-based Monte Carlo (T-Walk [112]), differential evolution (Diver [112]), and various grid, random and other toy samplers [112]. It also contains a dedicated postprocessor scanner plugin, which can be used for reprocessing samples obtained in a previous scan, either to recompute some output quantities or add new ones. See Ref. [112] for details.

When requesting a quantity in a scan, users are required to assign it a **purpose** in the context of that scan. The purpose may be `Observable` or `Test`, indicating that the quantity should be computed and output for every parameter combination sampled during a scan. Alternatively, a user can assign a `purpose` with a specific statistical meaning, such as `LogLike` or `Likelihood`. Interfaces to parameter sampling algorithms in ScannerBit allow the user to choose which `purpose` to associate with the objective function for the scanner at runtime. Following dependency resolution, GAMBIT creates a **likelihood container** from the module functions of the dependency tree that have been assigned the purpose(s) associated with the sampling algorithm. The likelihood container packages the module functions' combined results into a simple objective function for the sampler to call. The sampler then chooses parameter combinations to sample, sends each to the likelihood container, and receives the final likelihood for the parameter combination in return.

The GAMBIT convention is to assign `purpose:LogLike` to each component of a fit that is to be associated with the scanner, and for the module functions in question to return the natural log of the likelihood $\ln \mathcal{L}$. The likelihood container then combines the results of all such module functions by simply summing their return values, returning the result to the scanner as the total log-likelihood. All sampling algorithms interfaced in ScannerBit 1.0.0 allow only a single designated `purpose` to drive a scan, although other scanners to be connected in future versions will make use of multiple, different purposes within a single scan, for example to split likelihood calculations into 'fast' and 'slow' subsets [184].

8.2 Analysing samples

As it samples different parameter values, ScannerBit ensures that those values are output using whichever generalised print stream the user has selected (see Sect. 9 for details), along with all requested observables and likelihood components. The final task of statistical interpretation then requires parsing the printed samples and processing them into meaningful statistical quantities, whether Bayesian (posterior probability distribution functions, credible intervals and/or evidence ratios) or frequentist (profile likelihoods, confidence intervals and/or p values). Depending on the sampler employed, not all of these options may be valid (we return to this discussion in more detail in Ref. [112]).

Although processing the saved samples into statistical measures and producing corresponding plots are tasks technically outside the scope of GAMBIT itself, we specifically provide printer options that produce output compatible with common parsing and plotting software such as GetDist [185] and pippi [181]. We also provide a simple installer for pippi from within the GAMBIT integrated build system (Sect. 11). This allows GAMBIT to effectively produce profile likelihoods, confidence intervals, posterior probability distributions and maximum-posterior-density credible intervals *in situ*, by outsourcing the final step to pippi. Bayesian evidences can also be obtained directly from relevant scanners (e.g. MultiNest), or calculated after the fact with pippi. Calculating p values requires the user to make their own *ansatz* for the distribution of the GAMBIT log-likelihood (or other test statistic that they might choose to employ in a GAMBIT scan), and then convert the best fit identified by pippi to p. Future versions of ScannerBit are planned to include features designed to aid in determining this distribution.

8.3 Available likelihood forms

GAMBIT ships with a number of centrally-implemented, general purpose Gaussian and log-normal likelihood functions. These can be found in `Utils/src/statistics.cpp`. These are intended for use with simple observables and uncorrelated data, for implementing, e.g., nuisance likelihoods corresponding to well-measured SM parameters (see [111]). Module functions responsible for more complicated likelihood calculations typically contain their own implementations of appropriate test statistics, and corresponding translations to a quantity that can be treated as equivalent to $\ln \mathcal{L}$ in a scan (see the indirect detection likelihoods in [110], for example).

The centrally-implemented likelihoods come in a number of variants, allowing them to be used for detections, upper limits and lower limits. They deal with systematic uncertainties (theory errors, experimental systematics, related nuisance parameters, etc) by analytically profiling or marginal-

ising over an assumed distribution for an auxiliary parameter ϵ, which describes departures from a perfect mapping between model parameters and predicted values of observables. The module author must choose an appropriate central likelihood function to employ when computing a given likelihood. However, in every module function that uses one of following likelihoods, we choose to implement a boolean YAML option `profile_systematics` (default `false`) that selects at runtime whether systematics will be profiled or marginalised over.[21]

8.3.1 Profiled Gaussian

The basic Gaussian likelihood for data measured with some mean x and standard deviation σ, given a prediction μ, is

$$\mathcal{L}_G(x|\mu) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left[-\frac{1}{2}\frac{(x-\mu)^2}{\sigma^2}\right]. \quad (52)$$

Here μ may be a model parameter itself, or some complicated function of the true underlying model parameters. Taking ϵ to be an additive offset in μ induced by some source of error, and modelling its distribution as also Gaussian, centred on zero with standard deviation σ_ϵ, the joint likelihood becomes

$$\mathcal{L}_G = \frac{1}{2\pi\sigma\sigma_\epsilon} \exp\left[-\frac{1}{2}\frac{(x-\mu-\epsilon)^2}{\sigma^2} - \frac{1}{2}\frac{\epsilon^2}{\sigma_\epsilon^2}\right]. \quad (53)$$

Exactly how to denote $\mathcal{L}$ on the left of this equation depends on whether ϵ and σ_ϵ are to be interpreted to result from an auxiliary, independent measurement (frequentist), or simply some input systematic, possibly theoretical (Bayesian). In the former case, $\mathcal{L} = \mathcal{L}(x, \epsilon|\mu)$, and a final form of the likelihood for x alone can be obtained by profiling over the observed value of ϵ. To do this, we determine the value of ϵ that maximises $\mathcal{L}(x, \epsilon|\mu)$, by differentiating Eq. 53 to find the root

$$\hat{\epsilon} = \frac{\sigma_\epsilon^2}{\sigma^2 + \sigma_\epsilon^2}(x - \mu). \quad (54)$$

Substituting back into Eq. 53, the profiled version of the Gaussian likelihood is

$$\mathcal{L}_{G,\text{prof}}(x|\mu) = \frac{1}{2\pi\sigma\sigma_\epsilon} \exp\left[-\frac{1}{2}\frac{(x-\mu)^2}{\sigma^2 + \sigma_\epsilon^2}\right]. \quad (55)$$

---

[21] If the end user so desires, this can even be set differently for different module functions, although the resulting composite likelihood would arguably be inconsistent.

8.3.2 Marginalised Gaussian

If the quantity ϵ in Eq. 53 is instead interpreted as a direct input from e.g. theory, its Gaussian distribution has the character of a prior and $\mathcal{L} = \mathcal{L}(x|\mu, \epsilon, \sigma_\epsilon)$. Note that in this case, σ_ϵ has the character of a model parameter (or a quantity derived from the model parameters), indicating that it may vary as a function of the underlying model across the parameter space, independent of any considerations from data.

In this case, the appropriate likelihood for x alone instead comes from marginalising Eq. 53 over the possible values of ϵ, as

$$\mathcal{L}_{G,\text{marg}}(x|\mu, \sigma_\epsilon) = \frac{1}{2\pi\sigma\sigma_\epsilon} \int_{-\infty}^{\infty} e^{-\frac{(x-\mu-\epsilon)^2}{2\sigma^2} - \frac{\epsilon^2}{2\sigma_\epsilon^2}} \, d\epsilon, \quad (56)$$

giving

$$\mathcal{L}_{G,\text{marg}}(x|\mu, \sigma_\epsilon) = \frac{1}{\sqrt{2\pi(\sigma^2 + \sigma_\epsilon^2)}} \exp\left[-\frac{1}{2}\frac{(x-\mu)^2}{\sigma^2 + \sigma_\epsilon^2}\right]. \quad (57)$$

We compare the marginalised and profiled forms of the Gaussian likelihood for a toy problem with $x = 10$ and $\sigma = 1$ in the first panel of Fig. 6, assuming $\sigma_\epsilon = 0.5$ or $\sigma_\epsilon = 2$.

8.3.3 Profiled Gaussian limits

The simplest reasonable approximation to the underlying likelihood associated with an upper limit on an observable μ is to assume flatness below some canonical 'observed' or limiting value x, and to model the drop-off at $\mu > x$ with a Gaussian of width σ. This defines the piecewise function

$$\mathcal{L}_{G\uparrow}(x|\mu) = \begin{cases} \frac{1}{\sqrt{2\pi}\sigma}, & \text{if } \mu \leq x \\ \frac{1}{\sqrt{2\pi}\sigma} \exp\left[-\frac{1}{2}\frac{(x-\mu)^2}{\sigma^2}\right], & \text{if } \mu \geq x. \end{cases} \quad (58)$$

This treatment can be used to directly convert a measured value into a limit likelihood. An example is the relic density of dark matter $\Omega_\chi h^2$, which has been measured rather precisely, but may not consist exclusively of the dark matter candidate present in any particular BSM theory under investigation. The same treatment can also be used to implement likelihoods associated with published upper limits, but additional modelling is required to recover the equivalent central value x and falloff width σ from a published limit. Typically limits at two different CLs are needed to uniquely determine both x and σ.
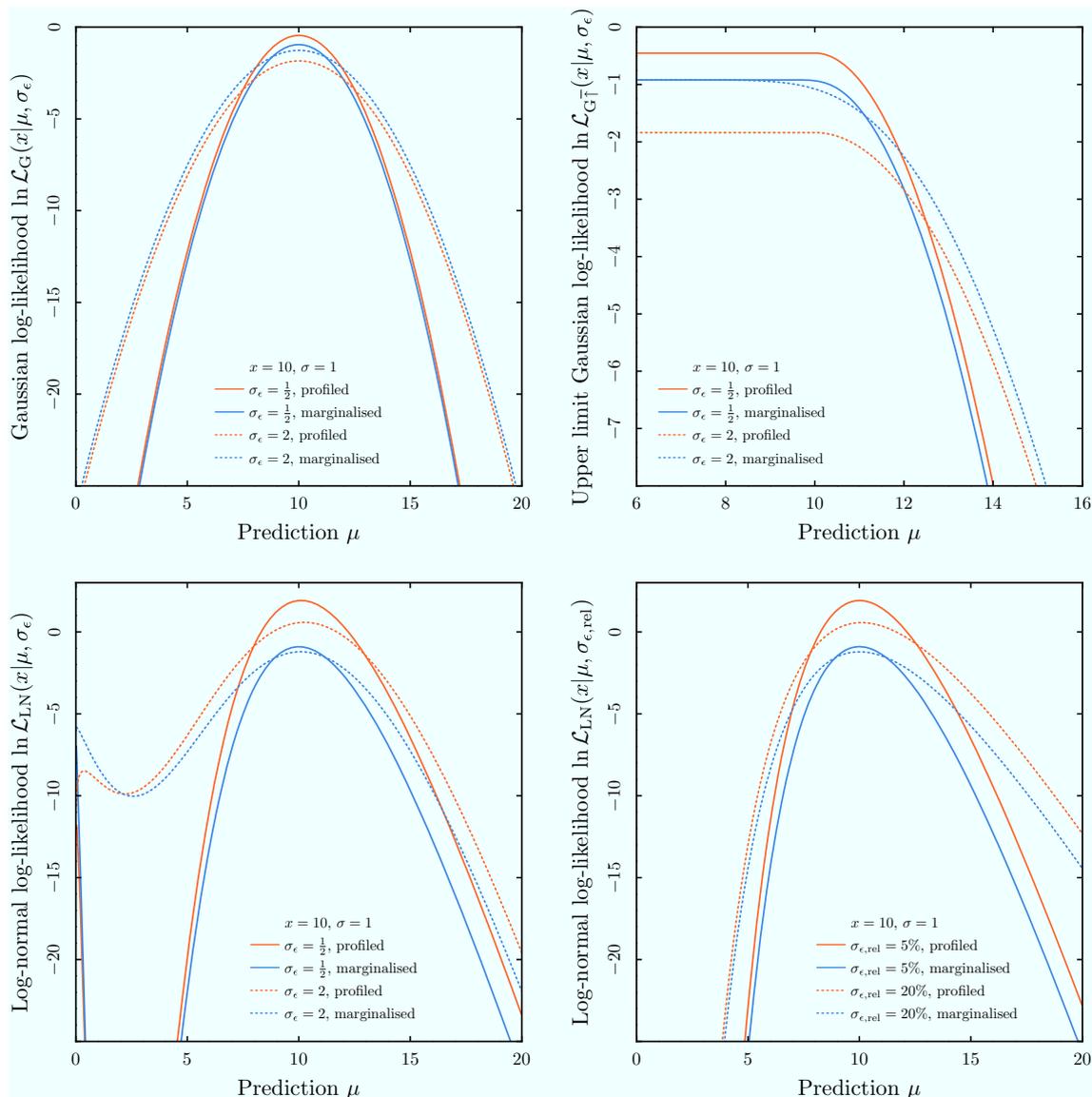
Fig. 6 The different generic likelihood functions available in **GAMBIT**, described in Sect. 8.3: Gaussian (*top left*), Gaussian limit (*top right*) and log-normal (*bottom*). Here we show the log-normal likelihood computed with a fixed *absolute* systematic uncertainty (*bottom left*), and instead with a fixed *fractional* (relative) systematic uncertainty (*bottom right*). Each curve is computed assuming an observed central value of $x = 10$ and standard deviation $\sigma = 1$, for two different assumed values of the systematic error. Two potential pitfalls are visible: the profiled upper limit likelihood shows a strong dependence on σ_ϵ at low values of μ, and adopting an absolute systematic uncertainty can introduce additional features in the log-normal likelihood at low μ

Including an uncertainty from some auxiliary nuisance observable ϵ proceeds similarly to the pure Gaussian case,

$$\mathcal{L}_{\overline{\text{G}\uparrow},\text{prof}}(x|\mu) = \frac{1}{2\pi\sigma\sigma_\epsilon} \max\left\{\max_{\epsilon \leq x-\mu} \exp\left[-\frac{\epsilon^2}{2\sigma_\epsilon^2}\right],\right.$$
$$\left.\times \max_{\epsilon \geq x-\mu} \exp\left[-\frac{(x-\mu-\epsilon)^2}{2\sigma^2} - \frac{\epsilon^2}{2\sigma_\epsilon^2}\right]\right\}. \quad (59)$$

Despite the need to carefully piecewise maximise in the different regimes, this leads to the simple result

$$\mathcal{L}_{\overline{\text{G}\uparrow},\text{prof}}(x|\mu) = \begin{cases} \frac{1}{2\pi\sigma\sigma_\epsilon}, & \text{if } \mu \leq x \\ \frac{1}{2\pi\sigma\sigma_\epsilon} \exp\left[-\frac{1}{2}\frac{(x-\mu)^2}{\sigma^2+\sigma_\epsilon^2}\right], & \text{if } \mu \geq x. \end{cases}$$
$$(60)$$

The corresponding expression $\mathcal{L}_{\text{G}\downarrow,\text{prof}}(x|\mu)$ for a lower limit is identical, except that the inequalities relating x and μ are reversed.

The simplicity of Eq. 60 is somewhat beguiling. Incorrectly using this expression when ϵ and σ_ϵ are interpreted in a Bayesian manner can lead to behaviour of the test statistic

that is undesirable in a frequentist analysis. For example, if σ_ϵ varies over the parameter space, the likelihood function will not actually be flat for $\mu \leq x$, despite the fact that the data make no statement about neighbouring values of μ in this region, and therefore neither should a sensible profile likelihood. An example of this behaviour can be seen in the second panel of Fig. 6. In such cases, it is important to carefully decide on the interpretation of ϵ and σ_ϵ from the outset. If they cannot be interpreted in a strict frequentist sense, then the marginalised variants of the likelihoods discussed here should be adopted instead, *even* when the final goal of a BSM scan is to produce profile likelihood results.

8.3.4 Marginalised Gaussian limits

To produce the marginalised form of Eq. 58, we again integrate the joint likelihood over all possible ϵ,

$$
\mathcal{L}_{G\bar{\uparrow},\text{marg}}(x|\mu,\sigma_\epsilon) = \frac{1}{2\pi\sigma\sigma_\epsilon} \int_{-\infty}^{x-\mu} \exp\left[-\frac{\epsilon^2}{2\sigma_\epsilon^2}\right] d\epsilon
$$
$$
+ \int_{x-\mu}^{\infty} \exp\left[-\frac{(x-\mu-\epsilon)^2}{2\sigma^2} - \frac{\epsilon^2}{2\sigma_\epsilon^2}\right] d\epsilon,
\tag{61}
$$

leading to

$$
\mathcal{L}_{G\bar{\uparrow},\text{marg}}(x|\mu,\sigma_\epsilon) = \frac{1}{2^{3/2}\sqrt{\pi}} \left[\frac{1}{\sqrt{\sigma^2+\sigma_\epsilon^2}} e^{-\frac{1}{2}\frac{(x-\mu)^2}{\sigma^2+\sigma_\epsilon^2}} \right.
$$
$$
\left. \times \text{erfc}\left(\frac{\sigma}{\sigma_\epsilon}\frac{x-\mu}{\sqrt{2(\sigma^2+\sigma_\epsilon^2)}}\right) + \frac{1}{\sigma}\text{erfc}\left(\frac{\mu-x}{\sqrt{2}\sigma_\epsilon}\right) \right], \tag{62}
$$

where $\text{erfc}(x) = \text{erf}(1-x)$ is the complementary error function. We can now see that

$$
\lim_{\mu\to-\infty} \mathcal{L}_{G\bar{\uparrow},\text{marg}}(x|\mu,\sigma_\epsilon) = \frac{1}{\sqrt{2\pi}\sigma}, \tag{63}
$$

regardless of σ_ϵ, and precisely as one would prefer a sensibly-behaved profile likelihood to do. This behaviour can be seen in the second panel of Fig. 6.

The corresponding marginalised likelihood for a lower limit $\mathcal{L}_{G\downarrow,\text{marg}}(x|\mu,\sigma_\epsilon)$ is obtained by making the replacements $x \to -x$ and $\mu \to -\mu$ in Eq. 62.

8.3.5 Profiled log-normal

A log-normal likelihood describes the situation where the distribution of the logarithm of some observation is expected to be Gaussian over repeated experiments. This may occur in cases where, for example, observations must return positive values by construction. The likelihood takes the form

$$
\mathcal{L}_{\text{LN}}(x|\mu) = \frac{1}{\sqrt{2\pi}\sigma'x} \exp\left[-\frac{1}{2}\frac{(\ln\frac{x}{\mu})^2}{\sigma'^2}\right]. \tag{64}
$$

Here x and μ remain the observed and predicted values of the observable, and the Gaussian distribution for $\ln x$ is centred on $\ln\mu$. The Gaussian width is σ', which is related to σ_{rel}, the relative uncertainty on x, as

$$
\sigma' \equiv \ln(1+\sigma_{\text{rel}}) = \ln(1+\sigma/x). \tag{65}
$$

This likelihood describes statistical variation in the scale of an observable, and is therefore most prone to the effects of systematics also able to impact that scale. In this case, ϵ takes the form of an auxiliary multiplicative source of error, with the corresponding additive offset given by $\ln\epsilon$. It is therefore appropriate to model the distribution of ϵ with a log-normal centred on 1. The corresponding width σ_ϵ' is then given by

$$
\sigma_\epsilon' \equiv \ln(1+\sigma_{\epsilon,\text{rel}}) = \ln(1+\sigma_\epsilon/\mu). \tag{66}
$$

The joint likelihood is then

$$
\mathcal{L}_{\text{LN}} = \frac{1}{\sqrt{2\pi}\sigma'\sigma_\epsilon'x\epsilon} \exp\left[-\frac{(\ln\frac{x}{\epsilon\mu})^2}{2\sigma'^2} - \frac{(\ln\epsilon)^2}{2\sigma_\epsilon'^2}\right]. \tag{67}
$$

This has its maximum at

$$
\hat{\epsilon} = \exp\left[\frac{\sigma_\epsilon'^2(\ln\frac{x}{\mu}-\sigma'^2)}{\sigma'^2+\sigma_\epsilon'^2}\right], \tag{68}
$$

leading to the profiled likelihood

$$
\mathcal{L}_{\text{LN,prof}}(x|\mu)
$$
$$
= \frac{1}{x\sqrt{2\pi}\sigma'\sigma_\epsilon'} \exp\left[-\frac{1}{2}\frac{\left(\ln\frac{x}{\epsilon\mu}\right)^2 + \sigma_\epsilon'^2\ln\frac{x}{\mu} - \sigma'^2\sigma_\epsilon'^2}{\sigma'^2+\sigma_\epsilon'^2}\right]. \tag{69}
$$

8.3.6 Marginalised log-normal

Integrating Eq. 67 over ϵ instead of maximising it gives the marginalised log-normal likelihood:

$$
\mathcal{L}_{\text{LN,marg}}(x|\mu) = \frac{1}{x\sqrt{2\pi(\sigma'^2+\sigma_\epsilon'^2)}} \exp\left[-\frac{1}{2}\frac{(\ln\frac{x}{\mu})^2}{\sigma'^2+\sigma_\epsilon'^2}\right]. \tag{70}
$$

In the lower panels of Fig. 6, we compare the marginalised and profiled forms of the log-normal likelihood, for the same toy problem as discussed previously ($x = 10$ and $\sigma = 1$). As in the Gaussian case, the profiled and marginalised versions show very similar behaviour, despite the fact that *unlike* the Gaussian case, they posses somewhat different functional

forms. Here we also show the additional features that can be induced at low μ if a constant value of the absolute systematic σ_ϵ is employed with the log-normal likelihood, rather than a constant relative uncertainty $\sigma_{\epsilon,\text{rel}}$.

9 Output

Output from GAMBIT scans is handled by the Printer subsystem, which generalises the writing of scan data to disk or any other output medium. It is designed so that the output format can be chosen at runtime with options in the master YAML file. Print commands within GAMBIT are issued via a general abstract interface, while the actual writing of data to the chosen output medium is handled by one of several derived classes, known as **printers**.

The actual print commands are automatically issued by GAMBIT whenever it runs a module function, so writers of new module functions need not concern themselves with how to send information to the printer system. Most users only need to know how to set up a printer via the master YAML file, and what the format of the output is. Section 6.6 covers the YAML setup. We deal with the main output format in Sect. 9.1, and the output formats of specific printers in Sect. 9.2.

There are three main scenarios that require additional knowledge of the printer system. One is writing a **scanner plugin**, where one must use the printer interface class to output e.g. probability weights or likelihoods. We discuss this briefly in Sect. 9.2.1, but we refer readers to the ScannerBit paper [112] for a full exposition. Another is when a user wishes to make an existing printer emit a new C++ type, to e.g. allow a new module function returning a custom type to print its result. We deal with this in Sect. 9.3. The final scenario is writing a new printer, for outputting GAMBIT data in a new format. This is a straightforward but quite specialised task, requiring complete knowledge of the class structure of the printer subsystem. The requisite details are left to documentation shipped with the code (found in `doc/writing_printers.pdf`)

9.1 Overview of the output format

Other than in scanner plugin code (see Ref. [112]), print commands are issued automatically to the GAMBIT printer system. This occurs after the evaluation of each module function that has been requested for printing from the master YAML file (see Sect. 6.6). Nonetheless, it useful to know how this system works when interpreting its output. The printer system receives print commands via functions with the signature

```
void _print(type const& result,
            const std::string& label,
            const int IDcode,
```

```
            const unsigned int MPIrank,
            const unsigned long pointID);
```

These contain the following information:

`result` The result computed by the module function (printable only if *type* is registered as printable and has an appropriate function overload defined; see Sect. 9.3).

`label` A string describing the `result` of the module function. It typically has the format

"#*capability* @*module*::*function*"

where *capability*, *module* and *function* are respectively the **capability**, host module and actual name of the module function that issued the print command. It is left to individual printers to decide what to do with this information (see Sects. 9.2.1, 9.2.2).

`IDcode` A unique integer automatically assigned to each module function for the duration of a scan. This allows printers to identify the origin of each print command without parsing the `label` string. Generally this number will not be passed on to the output file.

`MPIrank` The process rank assigned by MPI. Along with `pointID` this is needed to identify which parameter space point triggered a given print command.

`pointID` A unique integer automatically assigned to every parameter combination evaluated by a given MPI process in a scan. The `pointID` is not unique across processes, so both `MPIrank` and `pointID` need to be used in combination to obtain a globally unique identifier.

These arguments are the totality of information known to the printer at the time of each print command. It is then the job of the printer to assemble this information, from many print commands, into a coherent set of outputs.

Print commands can also be issued by ScannerBit or its plugins. By default, ScannerBit prints the final result returned to it for each model point (i.e. the total log-likelihood returned by the **likelihood container**).[22] However, scanners will often have other information that they want to record about each model point, and this can be added via manual calls to the print commands. Details can be found in the ScannerBit paper [112].

In addition to the module function, likelihoood container and scanner plugin outputs sent to the printer, the `MPIrank` and `pointID` are also automatically printed for every point. This allows printers the option of writing new data back to

---

[22] Technically, what is returned to the scanner is actually determined by the **purpose**(s) that the user has associated with their chosen scanner or test function plugin in the master YAML file (see Sect. 8). When using ScannerBit standalone however, *anything* can actually be connected to the scanner as its main objective function, and it will still be printed by default.

previous points. For example, the MultiNest scanner plugin computes posterior probability weights for a subset of points, long after the likelihood function is evaluated at those points. With this setup, such information can be inserted directly into the existing output medium at the position associated with those points, rather than having to write an entirely new output stream, as occurs in the native MultiNest output. It is up to the individual printers exactly how they handle this; for example, the ascii printer will write out a new file as MultiNest itself does, but the hdf5 printer will automatically update existing HDF5 files with new data about old points.

9.2 Available printers

Here we give specific details of how print commands are translated into files on disk by the ascii and hdf5 printers. This is essential information for interpreting the output of each printer.

9.2.1 ASCII output

The output file produced by the ascii printer (as named by the `output_file` option; see Sect. 6.6.1) consists of a simple whitespace-separated table of floating point numbers. The table is produced as follows. First, the GAMBIT module functions that are registered for printing issue print commands to the primary print stream, as they are each evaluated, and the result data is stored in a buffer. The print commands contain the `MPIrank` and `pointID` (see Sect. 9.1) identifying the model point that produced the data. By monitoring when these identifiers change, the printer detects when the scanner has moved to a new model point. Upon detecting a new point, the buffer begins a new line. Once the buffer is filled with a preset maximum number of lines, it is written to disk.

The structure of the ASCII output table (i.e. which data should be assigned to which column) is determined exclusively from the contents of the buffer immediately before the first dump. This imposes some extra restrictions on the data that the ascii printer can handle. For example, variable-length vectors of data can be printed, but at least one example with the maximum length expected in an entire scan must be sent to the printer before the first buffer dump, otherwise there will be insufficient space allocated in the output table to accommodate the longest such vectors in subsequent dumps.

To interpret the contents of the resulting ASCII file, an accompanying "info" file is produced at the time of the first buffer dump. The info file contains a list of labels identifying the columns of the output data file. If the data file is named `output.data`, then the info file will be `output.data_info`. When running GAMBIT via MPI, a separate output file will be produced for each process, with the rank of the host process appended to the root filename. An example info file

describing output generated by fitting a normal distribution with MultiNest [182][23] is shown below:

```
Column 1: unitCubeParameters[0]
Column 2: unitCubeParameters[1]
Column 3: MPIrank
Column 4: pointID
Column 5: LogLikelihood
Column 6: #NormalDist_parameters \
  @NormalDist::primary_parameters::mu
Column 7: #NormalDist_parameters \
  @NormalDist::primary_parameters::sigma
Column 8: #normaldist_loglike \
  @ExampleBit_A::normaldist_loglike
```

In this example the LogLikelihood (column 5) contains the global log-likelihood used to drive MultiNest. It consists of only one component, given in column 8: the log-likelihood returned by the normal distribution log-likelihood function `normaldist_loglike` from the module ExampleBit_A. Model parameter values are given in columns 6 and 7. The first two columns contain "unit hypercube" parameters, which are the raw unit-interval samples produced by MultiNest, before being transformed into the actual model parameter values by ScannerBit [112]. The `MPIrank` and `pointID` entries contain the model point identification data (Sect. 9.1).

Print statements originating from scanner plugins can be issued directly to the main printer – in which case they will be treated the same as module function output – or they can be issued to an auxiliary print stream if the data are not synchronised with the likelihood evaluations. Instructions for correctly handling this kind of data when writing scanner plugins are given elsewhere [112]. In the example above, unlike in the the native MultiNest output format, there are no posterior weights. These are issued to an auxiliary print stream in the MultiNest scanner plugin, so they end up in a different output file. The auxiliary file is also a plain ASCII table, and it comes with its own info file describing its contents:

```
Column 1: Posterior
Column 2: MPIrank
Column 3: pointID
```

The `Posterior` column contains the posterior weights, and the `MPIrank` and `pointID` contain the point identification data as before. Because `MPIrank` and `pointID` are shared between output files, they can be used to correlate Posterior weights with other data about the point during post-run analysis. GAMBIT could in principle perform this combination automatically at the end of a run, however it is currently left up to user's preferred post-run analysis tools. Note that the hdf5 printer *does* automatically combine the auxiliary print stream data with the primary print stream data, so it is the

---

[23] See the ScannerBit paper [112] for details of the GAMBIT interface to MultiNest.

more convenient format to use when working with auxiliary print data like posterior weights.

9.2.2 HDF5 output

The output file produced by the hdf5 printer is set with the `output_file` option (Sect. 6.6.3). It contains a separate data record for every output quantity, each of length equal to the number of parameter space points evaluated during the scan. These datasets are located according to the `group` option (Sect. 6.6.3). The command-line utility h5ls (included in most HDF5 library distributions) can be used to probe the internal layout of an HDF5 file. This can be useful for inspecting the names given to each dataset, which are derived from the `label` supplied via the print commands (see Sect. 9.1). The same information can also be obtained using the `probe` command in pippi [181].

All datasets in the resulting HDF5 files are synchronised, meaning that items at index i in every dataset have been obtained from the same model point. Each dataset comes with a second dataset of matching length, containing a flag indicating whether the data at the given index of the host dataset has been identified as valid or not. The labels for these datasets match their hosts, with `_isvalid` appended. For example, if an observable quantity `some_obs` was registered as invalid by GAMBIT for that point (perhaps because the result was unphysical), then the entry in the `some_obs_isvalid` dataset will be set to 0 (false). The `_isvalid` entries can thus be used as a mask for filtering out invalid or missing entries from the main dataset. This is done automatically in pippi— but a simple example Python script that uses h5py to inspect the HDF5 output of a GAMBIT run serves to illustrate the above concepts:

```python
import h5py
import numpy as np

#Open the hdf5 file
f = h5py.File("output_filename","r")

#Retrieve the log-likelihoods
logL_label = "group/LogLikelihood"
logL = f[logL_label]

#Retrieve flags indicating log-likelihood validity
isvalid_label = "group/LogLikelihood_isvalid"
mask = np.array(f[isvalid_label], dtype = np.bool)
print "Successful LogLikelihood evaluations:"
print np.sum(mask)

#Apply flags to print only valid log-likelihoods
print "Valid LogLikelihood values:"
print logL[mask]
```

Note that the output format described here applies only to the final, combined output of a GAMBIT scan. During a run, the information will be structured differently, and there will

be one output file for every MPI process involved in the scan. This output is combined when scans resume (so that new temporary output can be written), and when they complete. To examine the output of a scan in the format described here before the scan completes, it is necessary to stop the scan and then resume it (see Sect. 10.8) to trigger the combination.

9.3 Expanding the printable types

A module function result can only be printed if its C++ type is in the set of printable types. In order for a type to be printable, the printer chosen for a scan must have an appropriate print function overload defined for the type. The internal details of these function overloads must vary with the printer, as they describe how to translate the C++ type into the output format specific to each printer. Here we outline the general requirements.

The process is best illustrated with an example. Suppose one wishes to make the result type `std::map<std::string, int>` printable via the ascii printer. First, in order for the type to even be potentially printable by any printer, it must be listed in the `PRINTABLE_TYPES` macro in `Elements/include/gambit/Elements/printable_types.hpp`. Note that commas confuse the macro, so in this example the new type should first be aliased with a typedef, e.g.

```
typedef std::map<std::string,int> map_str_int
```

Next, one needs to add a new overload of the `print` function to the printer class (in this case the ascii printer). This requires a new declaration to be added to the class `asciiPrinter`. This can be acheived automatically by putting the type into one of the two lists of types to be found in `Printers/include/gambit/Printers/printers/asciitypes.hpp`:

```cpp
#define ASCII_TYPES                              \
  (std::string)                                  \
  // etc

#define ASCII_MODULE_BACKEND_TYPES               \
  (DM_nucleon_couplings)                         \
  (Flav_KstarMuMu_obs)                           \
  (map_str_int) // <--- New printable type.
```

Here the type should be added to ASCII_MODULE_BACKEND _TYPES if it is defined specifically as a module type or a backend type, and to ASCII_TYPES otherwise. Users unsure whether their type is a backend type, module type or some other type should be able to find the answer by studying Sects. 4.4, 10.5 and 11.1.

The corresponding function definition should then be added to `Printers/src/printers/asciiprinter/print_overloads.cpp`:

```cpp
void asciiPrinter::_print(map_str_int const&
 result, const std::string& label, const int
 IDcode, const unsigned int MPIrank, const
 unsigned long pointID)
{
  std::vector<std::string> names;
  std::vector<double> values;
  names.reserve( result.size() );
  values.reserve( result.size() );
  for (std::map<std::string, int>::iterator
   it = result.begin(); it != result.end(); it++)
  {
    std::stringstream ss;
    ss << label << "::" << it->first;
    names.push_back( ss.str() );
    values.push_back( it->second );
  }
  addtobuffer(values,names,IDcode,MPIrank,
   pointID);
}
```

Note that if the type appears in the `ASCII_TYPES` macro above, then the function definition should just go in the main body of `print_overloads.cpp`. If the type is instead part of `ASCII_MODULE_BACKEND_TYPES`, the function definition needs to be surrounded by the preprocessor directives `#ifndef SCANNER_STANDALONE ... #endif` in order to retain the ability to use ScannerBit without GAMBIT modules or backends.

Data can be supplied to the ascii printer buffer as a vector of values plus a matching vector of labels, so in this example the input string/integer map is simply converted into two vectors and sent to the print buffer. Of course, to fully understand the detail of the function body above one needs to understand the interior workings of the `asciiPrinter` class; those details can be found for each printer in the main code documentation (located in the `doc` directory).

In general, any expansion of the types printable by a given printer should also involve expanding the types *readable* by the corresponding 'inverse printer', which is used by the postprocessor scanner. See Ref. [112] for details.

10 Utilities

10.1 Particle database

The GAMBIT **particle database** provides standardised particle definitions for use throughout the code, in particular for referring to states in GAMBIT `Spectrum`, `DecayTable` and `ProcessCatalog` objects, which catalogue particle masses, decay and annihilation rates. It can be found in `Models/src/particle_database.cpp`.

Declaring new particles can be done either in singular form

```cpp
add_particle("~g", (1000021,0) )
```

or in sets, as

```cpp
add_particle_set("h0", ((25, 0), (35, 0)) )
```

In the first example, the gluino is declared with name `"~g"`, PDG code 1000021, and **context integer** 0. The context integer provides an additional index beyond the PDG code. This can be used to distinguish different particles that might employ the same PDG code under different circumstances, e.g. (s)fermion mass and gauge eigenstates.

In the second example, two new particles are declared, corresponding to the two neutral Higgses in the MSSM. The names of the new particles are constructed from the string `"h0"` and the indices of the individual particles in the set, such that `"h0_1"` is created with PDG code 25 and context zero, and `"h0_2"` is created with PDG code 35 and context zero. Essentially any number of particles can be placed together into a set in this manner.

Equivalent versions of both `add_particle` and `add_particle_set` exist for adding SM particles in particular; these are `add_SM_particle` and `add_SM_particle_set`. SM particles are given special treatment and saved as such inside the database, so that filters to e.g. decay final states can be applied according to whether one of the final states is or is not part of the SM.

A special version of `add_particle` also exists for defining broad particle classes like 'quarks', 'baryons', 'mesons', etc,

```cpp
add_generic_particle("quark", (9900084, 0))
```

These generic states have rather limited applicability, as they cannot participate in mass spectrum calculations, but they can prove useful for specifying generic decay channels.

Within the rest of GAMBIT, particles defined in the particle database can be referred to in three equivalent ways:

1. using their full name (e.g. `"~g"`, `"h0_1"`, `"h0_2"`, etc)
2. using their PDG-context integer pair (e.g. {35, 0})
3. using their short name and set index (e.g. {`"h0"`, 2})

The particle database itself contains various other helper functions for converting between these three conventions, and for converting particles into their corresponding antiparticles. It can be accessed using the function `ParticleDB()`, which returns a reference to the (singleton) database object.

The particle database in GAMBIT 1.0.0 contains entries for all SM and MSSM particles, as well as the singlet DM candidate, various significant SM mesons and generic particle classes.

10.2 Logging

The GAMBIT logging system provides a mechanism for writing important messages that occur during a run to disk, so that they can be examined when diagnosing problems or simply trying to understand a scan. Module writers can access the central logging singleton object via the accessor function `logger()`, which can be included via the header `Logs/include/gambit/Logs/log.hpp`. Log messages are sent to the logger object via the stream operator `<<`. Strings fed into the logger are concatenated until the special object `EOM` is received, which marks the end of each log message and causes it to be written to disk. A simple example is:

```
logger() << "Hello world" << EOM;
```

Log messages can be assigned tags depending on the nature and origin of the message. Tags can be used for automatically sorting log messages into different output files. For example, the tag `LogTags::info` can be attached to a message by inserting it via the stream operator at any point before the `EOM` object is received. A list of the available tags along with their string names used in the log output is given below:

```
// Message types
debug       = "Debug"
info        = "Info"
warn        = "Warning"
err         = "Error"
// Flags
fatal       = "Fatal"
nonfatal    = "Non-fatal"
// Component tags
def         = "Default"
core        = "Core"
logs        = "Logger"
models      = "Models"
dependency_resolver = "Dependency Resolver"
scanner     = "Scanner"
inifile     = "IniFile"
printers    = "Printers"
utils       = "Utilities"
backends    = "Backends"
```

Note that that namespace qualifier `LogTags` is required to access the tags.

If GAMBIT is compiled with MPI and run with more than one process, the MPI rank of the process that creates each log file is appended to its filename, separating log messages from different processes into different files.

By default, all log messages are delivered to the files runs/*yaml_filename*/logs/`default`.log_*rank*, where *yaml_filename* is the root name of the YAML file used to run GAMBIT, and *rank* is the MPI rank. There are two ways to change this default path. The first is to specify an override `default_output_path` in the `KeyValues` section of the YAML file (see Sect. 6.9). The second is to specify a new `prefix` in the `Logger` section of the YAML file, which specifies a directory in which to store log files, and overrides any

`default_output_path`. Log messages having a chosen set of tags can then be redirected into files in that directory using the `redirection` subsection under `Logger`. For example, to redirect all log messages to the files `new_def.log`_*rank* in the directory `/my_dir`, and all messages tagged with both `Error` and `Fatal` into the file `err_fatal.log`_*rank* in the same directory, the following `Logger` setup could be used:

```
Logger:
  prefix: "/my_dir/"
  redirection:
    [Default] : "new_def.log"
    [Error,Fatal] : "err_fatal.log"
```

The tag matching is inclusive, so any message containing the tags `Error` or `Fatal` will be directed to the file `err_fatal.log`, regardless of what other tags it also has. Such messages will also go to `new_def.log`, seeing as all messages have the `Default` tag.

By default, messages with the `Debug` tag will not be logged at all. The `Logger` option `debug` can be used to turn on debug log output, e.g.

```
Logger:
  debug: true
  redirection:
    [Debug] : "debug_log_messages.log"
```

The `Logger:debug` flag is automatically activated if the central `KeyValues:debug` flag is set true (cf. Sect. 6.9).

Messages delivered to the logger from within a **module** are automatically tagged with the name of the module, allowing messages originating from different modules to be easily isolated using the redirection system.

The log system does *not* capture regular print statements sent to `stdout` nor `stderr`. This means that any statements printed to the screen in **backends** or modules will appear in `stdout` and `stderr` as usual. This can be frustrating when working with a massively parallel MPI job. We advise users to take advantage of options for sending `stdout` and `stderr` to separate files for each MPI process, or tagging outputs with the MPI rank; these are available in the launcher applications (`mpiexec`, `mpirun`, etc) of essentially all MPI implementations, and in some batch schedulers as well.

10.3 Exceptions

GAMBIT has separate exceptions for errors, warnings and invalid parameter points, all of which derive from the C++ STL `exception` class. There is a single `invalid_point_exception` object created at initialisation for use throughout the code, along with a single `error` and a single `warning` object for each GAMBIT subsystem and each module. These are accessed by reference with the functions

```
invalid_point();

core_error();
dependency_resolver_error();
utils_error();
backend_error();
logging_error();
model_error();
Printers::printer_error();
IniParser::inifile_error();
DarkBit_error();
ScannerBit_error();
...

core_warning();
dependency_resolver_warning();
utils_warning();
backend_warning();
logging_warning();
model_warning();
Printers::printer_warning();
IniParser::inifile_warning();
DarkBit_warning();
ScannerBit_warning();
...
```

Flagging an invalid point is as simple as invoking the `raise` method with an appropriate explanation, e.g.

```
invalid_point().raise("Tachyon detected");
```

This causes the present parameter combination and the explanation to be logged, and the current module function evaluation to be terminated.

If an invalid point exception is raised during the calculation of the likelihood (or other `purpose` matching the scanner's requirements, cf. Sect. 8), it short-circuits the likelihood container. This causes all subsequent calculations in the dependency tree to be skipped, and the point declared invalid. In this way, by placing the module functions that are most likely to invalidate points earliest in the dependency tree, the dependency resolver can help to optimise a scan by preventing unnecessary calculations from being performed on points that turn out to be invalid for other reasons.

If the invalid point exception is raised during an observable calculation that is not needed for the likelihood, then the likelihood container simply notes that the calculated observable is invalid, and moves on to the next observable, without invalidating the actual likelihood value of the point.

Raising an error or a warning follows in a similar way to an invalid point, but also provides the possibility to provide an additional context string to facilitate future debugging, e.g.

```
DecayBit_error().raise(LOCAL_INFO,"Negative
 width!");
```

GAMBIT defines the macro `LOCAL_INFO` for this purpose, which unrolls to give a string with the exact file and line number in which it appears. Users can of course pass different context information if they prefer.

By default, errors are considered fatal and warnings non-fatal. Fatal exceptions cause a scan to terminate, printing the error message to `stdout`, whereas non-fatal ones are simply logged and the module function is allowed to continue. Invalid point exceptions, as well as errors and warnings set to be fatal, all eventually `throw` themselves in the manner of regular C++ exceptions; non-fatal errors and warnings never `throw`. Which errors and warnings are considered fatal can be modified from the `KeyValues` section of the input file, using options such as

```
exceptions:
  dependency_resolver_error: fatal
  dependency_resolver_warning: non-fatal
  core_warning: fatal
```

Sometimes, module writers will want to deliberately `raise` and then `catch` a GAMBIT exception. Invalid point exceptions always `throw` themselves, and if not caught earlier, are always caught and logged by the likelihood container. Module writers who wish to `raise` and `catch` invalid point exceptions within their own module functions can therefore safely do so using the regular `raise` function, under the understanding that any logging of the error is the responsibility of the catching routine.

The optional fatity of GAMBIT errors and warnings makes it impossible to do the same thing with them, however; despite being raised, an error or warning that is deemed non-fatal will never actually be thrown, let alone caught. GAMBIT errors and warnings therefore also provide `forced_throw` and `silent_forced_throw` methods as alternatives to `raise`. The `forced_throw` function raises and logs the exception as usual, but always throws it onwards, regardless of whether or not the exception is deemed fatal. The silent version does the same, but does no logging.

As throwing exceptions across OpenMP boundaries constitutes undefined behaviour, GAMBIT exceptions cannot be employed as usual from within nested module functions. To get around this problem, GAMBIT also includes global threadsafe deferred exception objects `piped_invalid_point`, `piped_errors` and `piped_warnings`. By calling `request` from these objects within a nested module function, an exception can be queued up for raising by the nested function's loop manager at the next opportunity. Developers of loop managers should therefore make a habit of calling `enquire` (inside OpenMP blocks) on the piped exception objects at regular intervals to see if any piped exceptions have been requested, and/or `check` (outside OpenMP blocks) to raise any queued exceptions.

BACKENDS	VERSION	PATH TO LIB	STATUS	#FUNC	#TYPES	#CTORS
DDCalc	1.0.0	Backends/installed/ddcalc/1.0.0/lib/libDDCalc.so	OK	36	0	0
DarkSUSY	5.1.3	Backends/installed/darksusy/5.1.3/lib/libdarksusy.so	OK	78	0	0
FeynHiggs	2.11.2	Backends/installed/feynhiggs/2.11.2/lib/libFH.so	absent/broken	14	0	0
	2.11.3	Backends/installed/feynhiggs/2.11.3/lib/libFH.so	OK	14	0	0
	2.12.0	Backends/installed/feynhiggs/2.12.0/lib/libFH.so	absent/broken	14	0	0
HiggsBounds	4.2.1	Backends/installed/higgsbounds/4.2.1/lib/libhiggsbounds.so	absent/broken	10	0	0
	4.3.1	Backends/installed/higgsbounds/4.3.1/lib/libhiggsbounds.so	OK	10	0	0
HiggsSignals	1.4	Backends/installed/higgssignals/1.4.0/lib/libhiggssignals.so	OK	12	0	0
LibFarrayTest	1.0	Backends/examples/libFarrayTest.so	OK	9	0	0
LibFirst	1.0	Backends/examples/libfirst.so	OK	8	0	0
	1.1	Backends/examples/libfirst.so	OK	15	0	0
LibFortran	1.0	Backends/examples/libfortran.so	OK	6	0	0
MicrOmegas_MSSM	3.6.9.2	Backends/installed/micromegas/3.6.9.2/MSSM/libmicromegas.so	OK	18	0	0
MicrOmegas_SingletDM	3.6.9.2	Backends/installed/micromegas/3.6.9.2/SingletDM/libmicromegas.so	OK	16	0	0
Pythia	8.212	Backends/installed/pythia/8.212/lib/libpythia8.so	OK	0	28	109
	8.212.EM	Backends/installed/pythia/8.212.EM/lib/libpythia8.so	absent/broken	0	28	109

Fig. 7 Example output of the backends diagnostic mode, showing the statuses and locations of different backend libraries configured for use with GAMBIT

10.4 Diagnostics

GAMBIT features extensive diagnostic tools, allowing users to quickly check which backends, scanners, modules and models are available at any given time, as well as which module and backend functions offer what capabilities for use in a scan.

10.4.1 Modules diagnostic

```
gambit modules
```

GAMBIT lists the modules present and available in the user's current configuration, indicating how many functions each module contains. Modules that are present on the user's system but have been excluded at configuration time from the compilation of GAMBIT are also listed, but are shown as ditched (see Sect. 11 for details on the `Ditch` process).

10.4.2 Capabilities diagnostic

```
gambit capabilities
```

GAMBIT lists all capabilities published by module functions, backend functions and backend variables, along with the modules and/or backends in which functions with each capability can be found.

10.4.3 Backends diagnostic

```
gambit backends
```

GAMBIT lists the backends for which it has frontend interfaces, by backend name and version. An example is shown in Fig. 7.

For each version of each backend, the diagnostic shows the path to the shared library containing the backend, the number of functions and variables published to GAMBIT by the frontend interface, the number of classes provided by the backend, and the number of different constructors it provides. The diagnostic also gives the overall status of the shared library of each backend. If the library has been located and loaded successfully, the status is OK; if it cannot be loaded or there was an error when loading it, the status is shown as absent/broken; if there was a problem finding the necessary symbols for any of the member functions of any of the classes provided by the backend, the status is shown as bad types.

In the case of a BOSSed library, any status other than OK causes GAMBIT to disable all module functions that are declared to need classes from that backend. Refer to the discussion of the rollcall declaration NEEDS_CLASSES_FROM in Sect. 4.5 for details.

Note that unlike constructor problems, symbol lookup errors for non-constructor backend functions or variables do *not* prevent a backend from presenting status OK overall. The status of individual functions and variables in a backend

backend_name can be probed using the free-form diagnostic `gambit` *backend_name*. Symbol errors from non-constructor backend functions and variables cause the individual functions/variables themselves to be disabled, but not the entire backend.

10.4.4 Models diagnostic

```
gambit models
```

GAMBIT lists the contents of the model database, giving the name of each model, its parent (if any), and the dimensionality of its parameter space. This diagnostic also produces the necessary files to generate a graph of the model hierarchy (e.g. Fig. 2).

10.4.5 Scanners diagnostic

```
gambit scanners
```

GAMBIT lists the names and versions of different parameter samplers for which it has interfaces defined in ScannerBit, and gives a status report on its efforts to load each of their shared libraries.

10.4.6 Test-functions diagnostic

```
gambit test-functions
```

GAMBIT lists the names and versions of different objective test functions known to ScannerBit, and gives a status report on its efforts to load each of their shared libraries.

10.4.7 Priors diagnostic

```
gambit priors
```

GAMBIT lists its known prior transformations for parameter sampling, giving a brief description of each along with the input file options that it accepts.

10.4.8 Free-form diagnostics

Further information can be found about essentially any component of GAMBIT by simply typing

```
gambit component
```

where *component* is a capability or the name of a module, backend, scanner plugin, test function plugin or model. The nature of the information subsequently provided depends on the type of component under investigation, but usually includes a short description, a status report and listings of the component's relationship to other components. Modules come with a list of the functions they contain, including function names, capabilities, types, dependencies and backend requirements. Backends come with similar information, as well as the individual status of each backend function. Scanners and test functions come with status reports, header and link info, and details of their accepted options. Models report detailed information about their family tree and the identities of their parameters. Asking about a capability generates a list of all module and backend functions able to compute that quantity.

10.5 Type handling

Dependency resolution works by matching module function capabilities and types to dependencies, and backend function capabilities and types to backend requirements. The types involved can be C++ intrinsic types, GAMBIT intrinsic types, or types associated specifically with a GAMBIT module, model or backend.

Types associated with specific backends are automatically made available to all GAMBIT modules and frontend routines, as all module functions can in principle have a backend requirement filled from any backend. In contrast, module types are used exclusively by functions associated with that module, and are not available to functions outside the module. The same is true of model types and their accessibility outside model-associated functions.

Adding a new type is relatively straightforward. General utility types that will be used throughout GAMBIT, and types with mixed backend, module and/or model character, should be declared in a new header, and that header included from `Utils/ include/gambit/Utils/shared_types.hpp`.

Backend types associated with backend *x* should be declared in a new header `Backends/include/ gambit/`↪`Backends/backend_types/`*x*`_types.hpp`. This header will then be automatically included in `Backends/ include/ gambit/Backends/backend_types_rollcall.hpp` by the GAMBIT build system (Sect. 11).

Types associated with model *y* should be declared in `Models/include/gambit/Models/models/`*y*`.hpp`, which will then also be picked up and included by the build system, this time in `Models/include/gambit/Models/ model_types_rollcall.hpp`.

Types for module *z* should be placed in a new header *z*`/include/gambit/`*z*`/`*z*`_types.hpp`, which must then be included from *z*`/include/gambit/`*z*`/`*z*`_rollcall.hpp`. The build system will automatically include the new header in `Elements/include/gambit/Elements/module_types _roll-call.hpp`.

The above discussion applies not only to new types, but also to typedefs associated with different components of GAMBIT. One challenge in performing dependency reso-

lution is that types are matched entirely as strings at run-time, meaning that the dependency resolver cannot recognise typedefs *a priori*. To allow it to understand typedefs and treat two types as equivalent for dependency resolution purposes, GAMBIT features a type equivalency database `config/resolution_type_equivalency_classes.yaml`. Entries in this file are equivalency classes of different types, such that the dependency resolver considers each type within an equivalency class to be identical to all the others in the same class.

10.6 Random numbers

Random numbers in GAMBIT are provided via a threadsafe wrapper to the random number generators of the C++11 STL `<random>`. Whether inside or outside an OpenMP block, single uniform random variates can be obtained by calling

```
double myran = Random::draw();
```

GAMBIT seeds and maintains a separate random number generator for each thread, so the resulting deviates are uncorrelated across threads. The seed for each generator is the sum of the system clock time and the thread index.

The underlying random number generator used by `Random::draw()` can be configured from the `KeyValues` section of the input file, as

```
KeyValues:
  rng: chosen_rng_engine
```

where *chosen_rng_engine* can be any of the recognised C++ random engines: `default_random_engine`, `minstd_rand`, `minstd_rand0`, `mt19937`, `mt19937_64`, `ranlux24_base`, `ranlux48_base`, `ranlux24`, `ranlux48` or `knuth_b`.[24] It can also be simply `default`, which selects the GAMBIT default generator; in GAMBIT 1.0.0 this is the 64-bit Mersenne Twister `mt19937_64`.

10.7 Component databases and documentation

Although this paper serves as a user and design guide to GAMBIT, as do Refs. [108–112] for each module, GAMBIT also features two additional documentation systems.

The first is a standard Doxygen documentation system, which gives basic information about classes, functions, variables, namespaces and macros defined in GAMBIT. The documentation can be generated with `make docs`, and is also available online at http://gambit.hepforge.org.

The second is a set of descriptive databases, which document individual models, capabilities, scanners, objective test functions and priors. These are the descriptions that are

brought up by the GAMBIT free-form diagnostic (cf. Sect. 10.4.8) when querying individual components with e.g.

```
gambit DarkBit
gambit NUHM1
gambit MultiNest
gambit SingletDM_spectrum
```

These can be edited or added to by modifying the text files

```
config/models.dat
config/capabilities.dat
config/scanners.dat
config/objectives.dat
config/priors.dat
```

These files are in fact written in YAML, albeit much simpler YAML than the input file. When adding a new model, scanner, test function or prior, or a module function with a capability that does not already exist in GAMBIT, it is good practice to also add a description of it to one of these files. If any component in GAMBIT is missing a description in these databases, a warning is raised whenever `gambit` is invoked.

10.8 Signal handling and resuming a scan

A GAMBIT scan can be terminated prematurely by sending it an appropriate POSIX signal, either SIGINT, SIGTERM, SIGUSR1, or SIGUSR2. Upon receiving one of these signals, GAMBIT will attempt to shut down cleanly, preserving output files and all information required for resuming the scan. The preservation of information required to resume scanning with a particular scan algorithm is the responsibility of ScannerBit, and more specifically each individual scanner plugin. This is described in detail in the ScannerBit paper [112].

To resume a scan, one simply re-launches GAMBIT using the same YAML file that was used to launch the original scan (making sure that the `-r` flag is not present in the argument list; if `-r` is present it will cause the scan to restart, deleting existing output). For example, a prematurely terminated scan that was launched using the YAML file *myrun.yaml* can be resumed simply by launching GAMBIT as:

```
gambit -f myrun.yaml
```

where `-f` indicates the input file to use (cf. Sect. 6.1).

10.8.1 Shutdown behaviour

There are two possible responses that GAMBIT might make when told to halt a run using a system (POSIX) signal. Which one it chooses depends on whether or not the scanner plugin in use can be instructed to stop by setting a `quit` flag. A scanner's ability to interpret a `quit` flag is automatically inferred by GAMBIT, on the basis of whether or not the scanner plugin calls `like_ptr::disable_external_shutdown()` in its constructor (see Appendix D of Ref. [112]).

---

[24] See http://www.cplusplus.com/reference/random/ for details.

First we discuss the case where the scanner can understand a `quit` flag. In this instance each GAMBIT MPI process will, upon receiving a shutdown signal, take the following actions:

1. Allow the current likelihood evaluation to complete as normal.
2. Broadcast a stop command via MPI to all other processes in the job. This triggers this same shutdown procedure in all other processes, and is necessary in case not all processes receive the original POSIX signal.
3. Finalise all printer output.
4. Set the `quit` flag for the scanner plugin.
5. Pass control back to the scanner plugin.

At this point the GAMBIT core system has completed its shutdown tasks, and assumes that the scanner plugin will do the same. The plugin should then complete its own shutdown tasks and return control to GAMBIT, which will then shut down MPI and exit the program.

The second case is where the scanner plugin has no way to recognise a `quit` flag. This is a less-than-ideal situation, as it makes performing a clean shutdown much more difficult, and indeed it is not possible to guarantee that shutdown will succeed in 100% of cases. However, some third-party scanning algorithms do not provide any mechanism to signal a premature end to a scan, so we have designed the shutdown system to work around this restriction.

In this case the shutdown procedure will operate as follows:

1. Allow the current likelihood evaluation to complete as normal.
2. Broadcast a stop command via MPI to all other processes in the job.
3. Enter a custom MPI barrier, and wait for for all other processes to signal that they have also entered the barrier.
4. If the synchronisation in 3 succeeds, skip to step 7.
5. If the synchronisation does not succeed within a set time interval, disable all future printer output and return control to the scanner plugin (which cannot be told to stop), and await the next likelihood evaluation request.
6. Upon being requested to evaluate another likelihood, immediately invalidate the model point and return to step 3.
7. Finalise all printer output.
8. Terminate the program.

The repeated synchronisation attempts are required because the scanner plugin may also be using MPI; because GAMBIT has no control over how MPI is used in third-party scanning algorithms, there is a high probability that a deadlock will occur between our synchronisation attempt and a blocking MPI command (from e.g. `MPI_Barrier`) in the third-

party code. We must therefore abandon our synchronisation attempt if takes too long, and return control to the scanning algorithm to allow deadlocks to resolve. However, more blocking calls can easily be initiated in some other process before we attempt to synchronise again, so we have to repeatedly attempt to find a window between these calls in which we can gain control over all MPI processes simultaneously. Once this succeeds, we can cleanly finalise the output, shut down MPI, and terminate the program.

It is possible that the synchronisation attempts never succeed. Because of this possibility, GAMBIT will only attempt the procedure a fixed number of times before giving up. In this case, each process will attempt to finalise its output and stop independently. In many cases this will succeed and there will be no problem. However, if a process has been left in a blocking call by a sampling algorithm, the process will fail to terminate, and will hang until killed by the operating system. This also has the potential to corrupt the printer output for that process. This is particularly true in the case of HDF5 output, as HDF5 files are highly vulnerable to corruption if not closed properly. This can result in data loss, and make a scan impossible to resume. Processes can also hang when running under certain MPI implementations if MPI is not finalised correctly, which cannot be done when processes are left to terminate independently.

We remind the reader that this second procedure is not the ideal case. The shutdown should always work smoothly if a `quit` flag can be set for the scanner plugin, so for large scans that produce valuable data we recommend using scanner plugins that have this feature (e.g. Diver; see [112] for details).

11 Configuration and automatic component registration

GAMBIT uses the open-source cross-platform build system CMake[25] to configure and build the package. The configuration step identifies the system architecture, available compilers, libraries and GAMBIT components present on a user's system, and creates appropriate makefiles. CMake and GAMBIT support in-source and out-of-source builds, but we recommend the latter for ease of organisation. The canonical way to configure and make GAMBIT is therefore

```
mkdir build
cd build
cmake ..
make
```

The build system also incorporates a series of Python **harvester scripts** for code generation. These are used at both configuration and compile time to automatically detect

---

modules, models, backends, printers, scanners, priors, test functions and associated types present in a user's personal GAMBIT configuration. The harvesters automatically write the header, configuration and CMake files required to register the various components in GAMBIT, and include them in the build. In this way, users wishing to add a GAMBIT component need only write the source/header files containing the component-specific content they wish to add, and place them in the appropriate folder, relying on the harvester scripts to generate the necessary boilerplate code required to incorporate the component into GAMBIT.

Compilation of GAMBIT module standalone executables is also handled by the same build system, but these are dealt with in Sect. 12.1.

11.1 Adding new models, modules, backends and other components to GAMBIT

Here we give a quick reference guide to the new files needed when adding new components, along with any notable modifications needed to existing files. Note that adding any new files to GAMBIT (or moving existing ones) necessitates re-running CMake before rebuilding.

modules Add a rollcall header *MyBit*/include/gambit /*MyBit*/*MyBit*_rollcall.hpp and source files in *MyBit* /include. See Sect. 3.1 for more details.

models Add a model declaration header Models/include/gambit/Models/models/*my_model*.hpp. If needed, add translation function source files to Models/src/ models. See Sect. 5.1 for more details. Adding a new model with no parent typically also requires either adding new module functions or verifying that existing ones are safe to use with the new model, and declaring them as such in their rollcall headers. See Sect. 3.1 for more details.

backends *We strongly encourage backend authors to go through the following simple steps, to provide official versions of the resulting GAMBIT interface files within the regular releases of their software.*

Add a frontend header Backends/include/gambit/ Backends/frontends/*backend_name_backend_version.* hpp. Add the default location of the backend's shared library to config/backend_locations.yaml.default. If needed, add backend convenience and/or initialisation function source files in Backends/src/frontends. If needed, add a backend type header in Backends/include /gambit/Backends/backend_types/ *backend_name* _types.hpp. If you want GAMBIT to manage the compilation of the backend, add an entry in cmake/ backends.cmake (see Sect. 11.2).

If only adding the latest *version* of an existing backend, then the new frontend header, any new frontend source

files, the new entry in config/backend_locations. yaml.default and any new entry in cmake/backends. cmake can usually just be copied from the previous versions and adapted. Be sure to update any #include statements in the copied source files to include the new frontend header rather than the previous one. Any new backend types, or modifications to old backend types, are generally best dealt with by adding the new and revised types to the existing backend types header, and declaring any revised types with entirely new names, in order to avoid breaking the frontend interface to previous versions of the backend.

printers Add a printer declaration header Printers/ include/gambit/Printers/printers/*my_printer*.hpp. If needed, add source files in Printers/src/printers/ *my_printer*. See Sect. 9 for more details.

scanners Add a scanner plugin declaration header Scanner Bit/include/gambit/ScannerBit/scanners/*scanner_ name*/*scanner_name*.hpp. Add any additional headers required to the same directory. If needed, add source files to ScannerBit/src/scanners/*scanner_name*. See Ref. [112] for more details. If you want GAMBIT to manage the compilation of the scanner, add an entry in cmake/ ↪scanners.cmake (see Sect. 11.2).

priors Add a prior declaration header in ScannerBit/ include/gambit/ScannerBit/priors. If needed, add source files in ScannerBit/src/priors. See Ref. [112] for more details.

objective test functions Add source files to ScannerBit/ src/objectives/test_functions. See Ref. [112] for more details.

types Exactly what to do depends on which component(s) the type is associated with; see the above entries and Sects. 4.4 and 10.5 for more information.

When adding any of these components, developers should also add a description of the new component to the relevant component database (see Sect. 10.7).

11.2 Building backends and scanners

Although not strictly necessary for running GAMBIT, we also provide helpful preconfigured methods within the configuration and build system for downloading, configuring, patching (where necessary) and compiling essentially all of the backends and scanners for which GAMBIT has frontend or scanner plugin interfaces. Although it is straightforward to just manually download and compile backends and scanners as usual, and then enter their shared library locations in custom config/backend_locations.yaml and config/scanner_locations.yaml files, using the automatic installers in the GAMBIT build system ensures that GAMBIT and all backends and scanners employ consistent

compiler and library settings. As with the C++ configuration of GAMBIT itself, for the compilation of backends written in C or Fortran, CMake automatically searches for the necessary compilers and libraries. Codes written in Python and other languages can only be backended by GAMBIT 1.0.0 if they ship with a C API; supporting 'native language' backending of such codes is a high priority for future versions.

This system provides a make target for each installed version of each scanner and backend, along with a corresponding make clean-*name* target (where *name* is the name of the backend or scanner), which calls distclean or similar in the backend source. Each scanner and backend also gets a target make nuke-*name*, which completely erases all downloaded and installed content for the component in question. The make scanners target installs and builds the latest versions of all registered external scanning algorithms, and the make backends target does the same for backends. All scanners or backends can be cleaned or nuked in one command with the make targets clean-scanners, nuke-scanners, clean-backends or nuke-backends. For the true nihilist, there is also nuke-all.

Adding a new backend or scanner to the GAMBIT automatic build management system is fairly straightforward. One adds a new 'external project' entry in either cmake/↪backends.cmake or cmake/scanners.cmake, using some of the built-in macros that can be seen demonstrated in those files, for setting up the clean targets and indicating if a given backend requires BOSSing. The minimum information required for a functional entry in either of these files is: the URL from which the package can be downloaded, the MD5 checksum of the download (obtainable for any file via cmake -E md5sum *filename*), and basic configure and build commands for the package. If required, specific build flags can be easily added to whatever GAMBIT passes to the backend. Custom patches can also be applied. If a backend should be BOSSed as part of the build process, a BOSS configuration file must be placed in the Backends/scripts/BOSS/configs directory, as described in Sect. 4.5. The configuration file should be named according to the backend name and safe version, e.g. MyBackend_1_2.py.

One important vagary of the build system for scanners: for GAMBIT to properly register a scanner as built and available, it is necessary to re-run cmake after making the external scanner, and then rebuild GAMBIT. The most efficient way to get GAMBIT started from scratch with e.g. MultiNest [182] or Diver [112] is therefore

```
mkdir build
cd build
cmake ..
make scanners
cmake ..
make
```

This particular requirement has its roots in the two-step shared library strategy that ScannerBit uses to dynamically load its plugins [112]. This will probably disappear in future versions.

11.3 Miscellaneous build targets

make get-pippi retrieves the latest development version of the analysis and plotting tool pippi [181] from GitHub[26], and places it in the GAMBIT root directory.

make docs builds the GAMBIT doxygen documentation.

make clean removes all compiled and automatically-generated source and header files for GAMBIT itself, but leaves backends and scanners untouched.

make distclean does the same as clean, but also cleans the GAMBIT doxygen documentation, clears out the GAMBIT scratch directory, and removes all downloaded, installed or compiled backend and scanner content.

11.4 Configuration options

Here we list the most useful commandline switches for passing to CMake when configuring GAMBIT, by

```
cmake -D OPTION_NAME = value
```

Often none of these is required, but they can be helpful for hinting or forcing CMake to use specific versions of compilers or libraries, or for simply disabling components or features of GAMBIT at the build stage.

11.4.1 *CMake standard variables*

CMAKE_BUILD_TYPE Sets the build type. Possible values are Debug, Release, Release_O3, MinSizeRel, RelWithDebInfo and None. The default is None, which results in the fastest build time, but no debug symbols and the slowest execution. Release includes optimisation seetings designed to result in the fastest run time; build time is correspondingly longer. Release_O3 is a GAMBIT-specific build type that differs from Release in that it passes -O3 rather than -O2 to the compiler.[27]

CMAKE_CXX_COMPILER Full path to the C++ compiler. Alternatively you can specify the environment variable CXX before invoking cmake.

CMAKE_C_COMPILER Full path to the C compiler. Alternatively you can specify the environment variable CC before invoking cmake.

---

[26] http://github.com/patscott/pippi.

[27] In this sense, Release_O3 in GAMBIT is actually closer to the traditional CMake definition of Release. It is not clear that Release_O3 offers any significant advantage over Release, however, and -O3 may cause instability in some backends. Use this option with caution.

`CMAKE_Fortran_COMPILER` Full path to the Fortran compiler. Alternatively you can specify the environment variable `FC` before invoking cmake.

`CMAKE_CXX_FLAGS` Extra flags to use when compiling C++ source files.

`CMAKE_C_FLAGS` Extra flags to use when compiling C source files.

`CMAKE_Fortran_FLAGS` Extra flags to use when compiling Fortran source files.

11.4.2 *CMake library and GAMBIT-specific variables*

`EIGEN3_INCLUDE_DIR` The full path to a local installation of Eigen. Note that Eigen can be installed automatically from many standard repositories, so a local installation may not be necessary.

`MPI` If `MPI=OFF`, MPI is manually disabled even if CMake successfully locates MPI libraries. Defaults to `ON`.

`MPI_INCLUDE` The full include path of the MPI distribution to be used (e.g. in case it is not detected automatically).

`MPI_LIBRARY` The full path to the MPI library file(s) to link against (e.g. in case they are not detected automatically).

`LAPACK_LINKLIBS` The full path to the LAPACK library file(s) to link against (e.g. in case they are not detected automatically).

`PYTHIA_OPT` If `PYTHIA_OPT=OFF` and the Intel compiler is in use, turn off cross-file interprocedural optimisation when compiling the BOSSed Pythia backend (some systems do not have enough memory to perform this optimisation step). Defaults to `ON`.

`Werror` If `True`, the build system treats all warnings as errors, and halts the build.

`(D)itch` Manually selects GAMBIT components to exclude from the build. Practically anything can be ditched with this command, from modules to models, backends, printers and scanners. The value should be set to a semicolon separated list of the beginnings of component names to match for ditching. For example,

```
cmake -Ditch = "Dark;FeynHiggs_2_11_";
```

would ditch the module DarkBit, all versions of the backend DarkSUSY, and versions 2.11.* of the FeynHiggs backend. Note that ditching a GAMBIT component does not 'clean' any compiled code, so it will not e.g. remove backend shared libraries that have already been compiled. It *will* however exclude all interfaces to the ditched components the next time GAMBIT is built, making it completely indifferent to the presence or absence of any compiled or uncompiled code associated with those components.

12 Examples, releases and support

12.1 Minimal examples

GAMBIT ships with a number of different examples. These include two minimal example modules (ExampleBit_A and ExampleBit_B) and some minimal backend examples in C (LibFirst) and Fortran (LibFortran and LibFarrayTest). A minimal toy model (NormalDist) can be found declared in `Models/include/gambit/Models/models/demo.hpp`. This file also contains a fully self-contained hierarchy of example models illustrating all the concepts of Sect. 5 (note that these are commented out by default, to avoid cluttering the model hierarchy). There is also a matching pair of example YAML files that use these modules and backends to run rudimentary scans of NormalDist (`yaml_files/spartan.yaml`) or the CMSSM (`yaml_files/spartan_CMSSM.yaml`). These two files each contain some simple additional entries, commented out by default, that can be used for experimenting with different printers and scanners. Most of the features and options outlined in this paper can be found demonstrated in one or another of these example components or scans.

The minimal `spartan` example also include a corresponding `pip` file (`yaml_files/spartan.pip`) for plotting the hdf5 results of `yaml_files/spartan.yaml` with pippi [181]. (This file will need to be altered if `yaml_files/spartan.yaml` is altered from its default.)

For more complete and realistic examples, users should refer to the full YAML files corresponding to the MSSM and scalar singlet scans described in Refs. [119–121], which also ship with GAMBIT and can be found in the `yaml_files` directory. These are `SingletDM.yaml`, `CMSSM.yaml`, `NUHM1.yaml`, `NUHM2.yaml` and `MSSM7.yaml`.

There are also a number of module-specific example YAML files to be found in the `yaml_files` folder: `WC.yaml` and `FlavBit_CMSSM.yaml` (for FlavBit), `ColliderBit_CMSSM.yaml` and `ColliderBit_ExternalModel.yaml` (for ColliderBit), `DecayBit_MSSM20.yaml` (for DecayBit), `PrecisionBit_MSSM20.yaml` (for PrecisionBit), `SpecBit_MSSM.yaml` and `SpecBit_vacuum_stability.yaml` (for SpecBit), `DarkBit_MSSM7.yaml` and `DarkBit_SingletDM.yaml` (for DarkBit), and `ScannerBit.yaml` (for ScannerBit).

The full GAMBIT distribution also includes a series of example driver programs that use the different modules as standalone libraries, without the rest of GAMBIT. Using GAMBIT modules in this manner requires some extra work due to the absence of the dependency resolver and related GAMBIT core components, but allows direct manual control of the functions in a given module, using only a minimal set of GAMBIT components. In certain cases, using GAMBIT modules as standalone libraries can be a lightweight and

even more flexible alternative to employing the full GAMBIT machinery.

The simplest standalone example is ExampleBit_A_standalone, found in `ExampleBit_A/examples/Example Bit_A _standalone.cpp`. Here the driver program carries out breezy versions of many of the tasks performed by GAMBIT in a full-blown scan. It first sets up some files to print log information to, and chooses which model to investigate. It identifies which module functions in ExampleBit_A it wants to run, then sets their internal options and connects their pipes to other functions within ExampleBit_A, and to relevant backends. It declares and defines an additional `QUICK_FUNCTION` (cf. Sect. 3.1.5) directly in the same file, to fill a missing dependency. It then chooses what parameter values to run through the resulting pipeline, gathers the results and prints them to `stdout`. Authors of standalone programs have the possibility to intervene in any of these steps, providing the necessary inputs from whatever source they like, or using the outputs directly in whichever manner they prefer.

For many models, the biggest challenge associated with using a module in standalone mode will be fulfilling dependencies on a GAMBIT `Spectrum` object, as these objects are typically created exclusively by SpecBit in a regular GAMBIT scan. For standalone purposes, each model with a `Spectrum` specialisation in SpecBit is expected to also have a correspondingly stripped-down **simple spectrum** defined in `Models/include/gambit/Models/SimpleSpectra`. The simple spectra are essentially mass container objects, devoid of any interface to an actual spectrum generator, which can be used in standalone executables in place of a true GAMBIT `Spectrum`. The GAMBIT `Spectrum` class and its simple spectrum variants are discussed in more detail in Ref. [111].

12.2 Releases

GAMBIT releases are assigned version numbers of the form `major.minor.revision`. Each version is available from the GAMBIT webpage: http://gambit.hepforge.org. The code can be downloaded either directly as a tarball, or accessed through a `git` repository, newly forked from the development branch at each minor version update.

As a convenience, for each release we also provide downloadable tarballs of each module, bundled with the minimal number of GAMBIT components required to use it in standalone mode. For physics modules, these components are the models, backend interfaces, logs and all other utilities except printers. The required components for ScannerBit are the printers, logs and a smaller subset of the utilities.

12.3 Support

Data used in GAMBIT observable and likelihood functions are generally available within regular releases of GAMBIT

or relevant backends. If in any future cases this is not possible for some reason, we will make them available from the main GAMBIT webpage.[28] Output samples from scans discussed in GAMBIT results papers (such as Refs. [119–121]) are also available from the main GAMBIT webpage.

General support information and relevant links are collected in the Support section of the GAMBIT webpage.[29] This includes the doxygen documentation for the latest GAMBIT release, a known issues page, and an FAQ dealing mostly with common configuration, compilation and backend questions.

GAMBIT will be supported with regular version updates and revisions. In general bug fixes will be applied in `revision` increments, and new features mostly in `minor` version increments. New `major` version increments will be reserved for substantial new features. After releasing a new `major` version, we will continue to support the last `minor` version of the superseded `major` version with bug fix updates (typically backported from the new `major`).

We welcome and encourage bug reports on GAMBIT. These should be submitted via the TRAC ticket system on the GAMBIT webpage.[30] To prevent spam, bug reporters will need to first sign up for an account with HEPforge.[31]

We also welcome enquiries from authors of existing or future backend codes about GAMBIT compatibility; we are willing to work with you to help optimise interoperability of your code with GAMBIT.

Users are also very welcome to suggest contributed code for release in a future version of GAMBIT, particularly new models, observables, likelihood functions, printers, scanners and backend interfaces. These suggestions will undergo a careful code review before being integrated into the main codebase. Submitters are expected to pay attention to the coding style of adjacent routines.

13 Summary

GAMBIT is a powerful, general, flexible and extensible tool for phenomenological and statistical analysis of particle theories Beyond the Standard Model. It includes modules specialised for spectrum and decay calculations, collider, flavour, DM and precision physics, a hierarchical model database of popular BSM theories, flexible interfaces to many of the most popular existing phenomenology codes, extensive statistical and parameter scanning options, and an automatic system for connecting different calculations to their required inputs, outputs and models. Here we have outlined

---

[28] http://gambit.hepforge.org.

[29] http://gambit.hepforge.org/support.

[30] http://gambit.hepforge.org/trac/report.

[31] https://www.hepforge.org/register.

the main features of the GAMBIT package itself; accompanying papers lay out the details of the individual modules [108–112] and present first BSM results [119–121]. The package is fully open source, and can be downloaded from gambit.hepforge.org.

Appendix A: Quick start guide

To configure and build GAMBIT on a machine with n logical cores, retrieve the git repository or the tarball and unpack it, then

```
cd gambit
mkdir build
cd build
cmake ..
make -jn scanners
cmake ..
make -jn gambit
```

To build all backends supported for automatic download:

```
make -jn backends
```

You can also build individual backends with

```
make -jn backend_name
```

and clean them with

```
make clean-backend_name.
```

To see which backends and scanners have been installed correctly, do

```
gambit backends
```

and

```
gambit scanners
```

To run gambit using the included example YAML files `spartan.yaml` or `MSSM7.yaml`, do

```
gambit -f yaml_files/spartan.yaml
gambit -f yaml_files/MSSM7.yaml
```

To make a standalone example using any one of the modules:

```
make module_name_standalone
```

and run the resulting executable *module_name* `_standalone`.

For more details on the configuration and build options, please see Sects. 11.2–11.4.

Appendix B: Supported compilers and library dependencies

GAMBIT builds and runs under Linux and Mac OS X; the architecture is automatically detected by the build system.

GAMBIT is written in C++ and requires a compiler that supports a minimal subset of the ISO C++11 standard. For compiling Fortran backends, a Fortran compiler is also required. GAMBIT supports GNU[32] and Intel[33] C/C++ and Fortran compilers. The Clang[34] C/C++ compiler is not supported due to its lack of historical support for OpenMP. When newer versions supporting OpenMP become the default in OS X, we will add support for clang in GAMBIT.

The following prerequisite libraries and packages must be installed to configure and to build GAMBIT:

– gcc/gfortran 4.7.1 or greater, or icc/ifort 12.1.0 or greater
– Python 2.7 or greater (Python 3 is not supported)
– Python modules yaml, os, re, datetime, sys, getopt, shutil and itertools. Also h5py if using the hdf5 printer with pippi.
– Boost[35] 1.41 or greater
– GNU Scientific Library (GSL)[36] 1.10 or greater

---

[32] https://gcc.gnu.org/.
[33] https://software.intel.com/en-us/intel-compilers.
[34] http://clang.llvm.org/.
[35] http://www.boost.org/.
[36] http://www.gnu.org/software/gsl/.

Two additional linear algebra libraries are also currently required, but will become optional in future releases (where FlexibleSUSY will become a full backend, rather than shipping in the GAMBIT `contrib` directory):

– Eigen[37] 3.1.0 or greater: required only if using FlexibleSUSY from SpecBit or GM2Calc from PrecisionBit
– LAPACK[38]: required only if using FlexibleSUSY from SpecBit or MultiNest from ScannerBit

The following are optional libraries and packages:

– MPI: required only if parallelised sampling is desired
– axel[39]: if available, will be used to speed up downloads of backends and scanners wherever possible
– graphviz[40]: required only if model hierarchy and dependency tree plots are desired
– HDF5[41]: required only if using the hdf5 printer
– ROOT[42] 5.*: required if using Delphes from ColliderBit, or GreAT from ScannerBit

If any optional package is missing, the build system automatically `-Ditches` the corresponding component or feature relying on the missing package.

Users should note that whilst GAMBIT itself compiles and runs with a wide range of compiler versions, some backend and scanner codes are not compatible with certain newer compilers. A continually-evolving list of compiler versions tested to date with different backends can be found at http://gambit.hepforge.org/compilers. Whilst we obviously cannot assume responsibility for the portability of codes maintained by other members of the community, we are actively working with the authors of the different codes to help improve this situation.

Appendix C: Standard Model definitions

The parameters of the **StandardModel_SLHA2** model are:

`CKM_A` (A): Wolfenstein parameter defined in $\overline{MS}$ scheme at scale m_Z. Converted into V_{CKM} entries using the 9th-order expansions of Ref. [32].

`CKM_etabar` ($\bar{\eta}$): Wolfenstein parameter defined in $\overline{MS}$ scheme at scale m_Z. Converted into V_{CKM} entries using the 9th-order expansions of Ref. [32].

`CKM_lambda` (λ): Wolfenstein parameter defined in $\overline{MS}$ scheme at scale m_Z. Converted into V_{CKM} entries using the 9th-order expansions of Ref. [32].

`CKM_rhobar` ($\bar{\rho}$): Wolfenstein parameter defined in $\overline{MS}$ scheme at scale m_Z. Converted into V_{CKM} entries using the 9th-order expansions of Ref. [32].

`GF` (G_F): Fermi coupling, in units of GeV^{-2}.

`alpha1` (α_1): First Majorana CP-violating phase of the PMNS matrix, in radians.

`alpha2` (α_2): Second Majorana CP-violating phase of the PMNS matrix, in radians.

`alphaS` ($\alpha_s(m_Z)^{\overline{MS}}$): Strong coupling in $\overline{MS}$ scheme at scale m_Z.

`alphainv` ($\alpha_{\text{EM}}^{-1}(m_Z)^{\overline{MS}}$): Inverse electromagetic coupling in 5-flavour $\overline{MS}$ scheme at scale m_Z.

`delta13` (δ_{13}): Majorana CP-violating phase of the PMNS matrix, in radians.

`mBmB` ($m_b(m_b)^{\overline{MS}}$): $\overline{MS}$ mass of the b quark at scale m_b, in GeV.

`mCmC` ($m_c(m_c)^{\overline{MS}}$): $\overline{MS}$ mass of the c quark at scale m_c, in GeV.

`mD` ($m_d(2\,\text{GeV})^{\overline{MS}}$): $\overline{MS}$ mass of the d quark at scale of 2 GeV, in GeV.

`mE` (m_e): Pole mass of the electron, in GeV.

`mMu` (m_μ): Pole mass of the muon, in GeV.

`mNu1` ($m_{\tilde{\nu}_1}$): Pole mass of first left-handed neutrino mass eigenstate, in GeV.

`mNu2` ($m_{\tilde{\nu}_2}$): Pole mass of second left-handed neutrino mass eigenstate, in GeV.

`mNu3` ($m_{\tilde{\nu}_3}$): Pole mass of third left-handed neutrino mass eigenstate, in GeV.

`mS` ($m_s(2\,\text{GeV})^{\overline{MS}}$): $\overline{MS}$ mass of the s quark at scale of 2 GeV, in GeV.

`mT` (m_t): Pole mass of the t quark, in GeV.

`mTau` (m_τ): Pole mass of the τ lepton, in GeV.

`mU` ($m_u(2\,\text{GeV})^{\overline{MS}}$): $\overline{MS}$ mass of the u quark at scale of 2 GeV, in GeV.

`mZ` (m_Z): Pole mass of the Z boson, in GeV.

`theta12` (θ_{12}): Solar neutrino mixing angle of the PMNS matrix, in radians.

`theta23` (θ_{23}): Atmospheric neutrino mixing angle of the PMNS matrix, in radians.

`theta12` (θ_{13}): Reactor neutrino mixing angle of the PMNS matrix, in radians.

Appendix D: Glossary

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

---

[37] http://eigen.tuxfamily.org.

[38] http://www.netlib.org/lapack/.

[39] http://axel.alioth.debian.org/.

[40] http://www.graphviz.org/.

[41] https://www.hdfgroup.org/HDF5/.

[42] https://root.cern.ch/.

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

backend convenience function A function constructed purely from calls to **backend functions** and/or using **backend variables** from a single **backend**, wrapped by the backend's **frontend** in such a way as to appear to GAMBIT as just another function in that backend. It calculates a specific quantity indicated by its **capability**. Its capability and call signature are defined in the backend's **frontend header**.

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 initialisation function A function that automatically runs for each parameter combination, before any functions or variables from a given backend are used. It is defined as part of the backend's **frontend** interface. Although they are each associated with a specific backend, backend initialisation functions are actually technically **module functions** belonging to a system-defined module called BackendIniBit.

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).

child model A model that descends from a **parent model**, implying that any point in the parameter space of the child model can be expressed as a physically-equivalent point in the parent model's parameter space.

conditional dependency A **dependency** that only applies under specific circumstance, such as when a particular model or **backend** is in use.

context integer An integer assigned to a particular particle definition in the GAMBIT **particle database**, in order to distinguish it from another similar particle having the same PDG code.

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

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

friend model A model into which points from another model can be translated, even though the friend model is not a direct ancestor of the other model.

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.

harvester script One of a set of Python scripts that runs during GAMBIT configuration, and harvests information about the **modules**, **models**, **backends**, printers, scanners, priors, test functions and associated **types** present in the local installation of GAMBIT.

likelihood container The interface between ScannerBit and the graph of **module functions** created by the **dependency resolver**. It returns the total combined likelihood for any given set of model parameter values.

loop manager A type of **module function**, able to run **nested module functions** in parallel using OpenMP, from within its own function body.

model A GAMBIT model is defined as a collection of named parameters, intended for sampling by a scanning algorithm according to some prior. The scanner and prior are both chosen at runtime.

model group A set of **models** defined for easy reference when setting rules about what combinations of models are required for using a given **module function**. A model group is declared within the declaration of a module function in a **rollcall header**.

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.

printer The main object handling GAMBIT output. Multiple versions of this object exist (and new ones can be written), for handling output to different formats. Users select which printer they want to use via the master initialisation file (Sect. 6.6)

purpose A tag attached to a request made by a user in the `ObsLikes` section of their YAML file. The tag is used by the scanner and **likelihood container** to select which module functions to include in the combined likelihood and use for directing the scan.

nested module function A **module function** that must be run by a **loop manager** rather than directly by GAMBIT itself — usually in parallel inside an OpenMP block managed by the loop manager.

parent model A model from which a **child model** descends, implying that any point in the parameter space of the child model can be interpreted as a point in (some subspace of) the parent parameter space.

particle database An internal database in GAMBIT containing the names and PDG codes of all particles recognised by GAMBIT.

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

pipe A pointer to another function or variable created for allowing a specific **module function** to exchange information with other parts of the code. The pointer is set at runtime by the **dependency resolver**.

quantity This term is often used as short-hand for the combination of a **capability** with a certain **type**.

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

rule A directive given in the input YAML file that specifies options for one or more **module functions** and/or constraints on how the functions' **dependencies** or **backend requirements** may be resolved by the **dependency resolver**.

safe version A backend version number, but with all periods replaced by underscores (so as to be usable in automatically-generated namespaces, variables names, etc).

scanner plugin An interface in ScannerBit to an external code for parameter sampling, i.e. a scanner.

simple spectrum A minimal GAMBIT Spectrum object, designed to simply act as a container for pole masses and other spectrum data. Unlike a fully-fledged Spectrum object, it specifically does not provide RGE functionality or an interface to a spectrum generator. Designed to facilitate basic standalone use of modules.

test function plugin An interface in ScannerBit to a test function, which may be used for testing purposes as the objective function for a scan, in place of the output from the **likelihood container**.

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

References

1. ATLAS Collaboration, Search for resonances decaying to photon pairs in 3.2 fb^{-1} of pp collisions at $\sqrt{s} = 13$ TeV with the ATLAS detector. ATLAS-CONF-2015-081 (2015)
2. ATLAS Collaboration, Summary of the ATLAS experiment's sensitivity to supersymmetry after LHC Run 1—interpreted in the phenomenological MSSM. JHEP **10**, 134 (2015). arXiv:1508.06608
3. CMS Collaboration, Search for supersymmetry in the multijet and missing transverse momentum final state in pp collisions at 13 TeV. Phys. Lett. B **758**, 152–180 (2016). arXiv:1602.06581
4. 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
5. T. Abe, I. Adachi et al., Belle II technical design report. arXiv:1011.0352
6. CMS and LHCb Collaborations, Observation of the rare $B_s^0 \to \mu^+\mu^-$ decay from the combined analysis of CMS and LHCb data. Nature **522**, 68–72 (2015). arXiv:1411.4413
7. XENON100 Collaboration, E. Aprile, M. Alfonsi et al., Dark matter Results from 225 live days of XENON100 data. Phys. Rev. Lett. **109**, 181301 (2012). arXiv:1207.5988
8. C. Amole, M. Ardid et al., Dark matter search results from the PICO-60 CF$_3$ I bubble chamber. Phys. Rev. D **93**, 052014 (2016). arXiv:1510.07754
9. D.S. Akerib, H.M. Araújo et al., Improved limits on scattering of weakly interacting massive particles from reanalysis of 2013 LUX data. Phys. Rev. Lett. **116**, 161301 (2016). arXiv:1512.03506
10. Planck Collaboration, P.A.R. Ade et al., Planck 2015 results. XIII. cosmological parameters. A&A **594**, A13 (2016). arXiv:1502.01589
11. T.R. Slatyer, Indirect dark matter signatures in the cosmic dark ages. I. Generalizing the bound on s-wave dark matter annihilation from Planck results. Phys. Rev. D **93**, 023527 (2016). arXiv:1506.03811
12. R. Adhikari, M. Agostini et al., A white paper on keV sterile neutrino dark matter. arXiv:1602.04816
13. T. Bringmann, C. Weniger, Gamma ray signals from dark matter: concepts, status and prospects. Phys. Dark Univ. **1**, 194–217 (2012). arXiv:1208.5481
14. *Fermi*-LAT Collaboration, M. Ackermann, A. Albert et al., Searching for dark matter annihilation from milky way dwarf spheroidal galaxies with six years of *Fermi* large area telescope data. Phys. Rev. Lett. **115**, 231301 (2015). arXiv:1503.02641
15. IceCube Collaboration, M.G. Aartsen et al., Improved limits on dark matter annihilation in the Sun with the 79-string IceCube detector and implications for supersymmetry. JCAP **04**, 022 (2016). arXiv:1601.00653
16. DAMA Collaboration, R. Bernabei, P. Belli et al., First results from DAMA/LIBRA and the combined results with DAMA/NaI. Eur. Phys. J. C 167 (2008). arXiv:0804.2741
17. L. Goodenough, D. Hooper, Possible evidence for dark matter annihilation in the inner milky way from the *Fermi* gamma ray space telescope. arXiv:0910.2998
18. O. Adriani, G.C. Barbarino et al., An anomalous positron abundance in cosmic rays with energies 1.5–100 GeV. Nature **458**, 607–609 (2009). arXiv:0810.4995

19. CoGeNT Collaboration, C.E. Aalseth, P.S. Barbeau et al., Search for an annual modulation in a p-type point contact germanium dark matter detector. Phys. Rev. Lett. **107**, 141301 (2011). arXiv:1106.0650

20. T. Bringmann, X. Huang, A. Ibarra, S. Vogl, C. Weniger, Fermi-LAT search for internal bremsstrahlung signatures from dark matter annihilation. JCAP **7**, 54 (2012). arXiv:1203.1312

21. E. Bulbul, M. Markevitch et al., Detection of an unidentified emission line in the stacked X-ray spectrum of galaxy clusters. ApJ **789**, 13 (2014). arXiv:1402.2301

22. A. Boyarsky, O. Ruchayskiy, D. Iakubovskyi, J. Franse, Unidentified line in X-ray spectra of the andromeda galaxy and Perseus galaxy cluster. Phys. Rev. Lett. **113**, 251301 (2014). arXiv:1402.4119

23. A.C. Vincent, P. Scott, A. Serenelli, Possible indication of momentum-dependent asymmetric dark matter in the sun. Phys. Rev. Lett. **114**, 081302 (2015). arXiv:1411.6626

24. ATLAS Collaboration, G. Aad, B. Abbott et al., Search for high-mass diboson resonances with boson-tagged jets in proton-proton collisions at $\sqrt{s} = 8$ TeV with the ATLAS detector. JHEP **12**, 55 (2015). arXiv:1506.00962

25. CDMS Collaboration, Z. Ahmed et al., Search for annual modulation in low-energy CDMS-II data. arXiv:1203.1309

26. R. Bartels, S. Krishnamurthy, C. Weniger, Strong support for the millisecond pulsar origin of the Galactic center GeV excess. Phys. Rev. Lett. **116**, 051102 (2015). arXiv:1506.05104

27. S.K. Lee, M. Lisanti, B.R. Safdi, T.R. Slatyer, W. Xue, Evidence for unresolved γ-ray point sources in the inner galaxy. Phys. Rev. Lett. **116**, 051103 (2016). arXiv:1506.05124

28. T. Jeltema, S. Profumo, Deep XMM observations of Draco rule out at the 99 per cent confidence level a dark matter decay origin for the 3.5 keV line. MNRAS **458**, 3592–3596 (2016). arXiv:1512.01239

29. G. Angloher, A. Bento et al., Limits on momentum-dependent asymmetric dark matter with CRESST-II. Phys. Rev. Lett. **117**, 021303 (2016). arXiv:1601.04447

30. A.B. Arbuzov, M. Awramik, M. Czakon et al., ZFITTER: a semi-analytical program for fermion pair production in e^+e^- annihilation, from version 6.21 to version 6.42. Comput. Phys. Commun. **174**, 728–758 (2006). arXiv:hep-ph/0507146

31. M. Baak, M. Goebel et al., Updated status of the global electroweak fit and constraints on new physics. Eur. Phys. J. C **72**, 2003 (2012). arXiv:1107.0975

32. J. Charles, A. Höcker et al., CP violation and the CKM matrix: assessing the impact of the asymmetric B factories. Eur. Phys. J. C **41**, 1–131 (2005). arXiv:hep-ph/0406184

33. F. Capozzi, G.L. Fogli et al., Status of three-neutrino oscillation parameters, circa 2013. Phys. Rev. D **89**, 093018 (2014). arXiv:1312.2878

34. D.V. Forero, M. Tórtola, J.W.F. Valle, Neutrino oscillations refitted. Phys. Rev. D **90**, 093006 (2014). arXiv:1405.7540

35. J. Bergström, M.C. Gonzalez-Garcia, M. Maltoni, T. Schwetz, Bayesian global analysis of neutrino oscillation data. JHEP **9**, 200 (2015). arXiv:1507.04366

36. E.A. Baltz, P. Gondolo, Markov chain Monte Carlo exploration of minimal supergravity with implications for dark matter. JHEP **10**, 52 (2004). arXiv:hep-ph/0407039

37. B.C. Allanach, C.G. Lester, Multidimensional mSUGRA likelihood maps. Phys. Rev. D **73**, 015013 (2006). arXiv:hep-ph/0507283

38. R. Lafaye, T. Plehn, D. Zerwas, SFITTER: SUSY parameter analysis at LHC and LC. arXiv:hep-ph/0404282

39. R. Ruiz de Austri, R. Trotta, L. Roszkowski, A Markov chain Monte Carlo analysis of CMSSM. JHEP **5**, 2 (2006). arXiv:hep-ph/0602028

40. R. Trotta, R.R. de Austri, L. Roszkowski, Prospects for direct dark matter detection in the constrained MSSM. New Astron. Rev. **51**, 316–320 (2007). arXiv:astro-ph/0609126

41. L. Roszkowski, R. Ruiz de Austri, R. Trotta, Implications for the constrained MSSM from a new prediction for b → sγ. JHEP **7**, 75 (2007). arXiv:0705.2012

42. L. Roszkowski, R. Ruiz de Austri, J. Silk, R. Trotta, On prospects for dark matter indirect detection in the constrained MSSM. Phys. Lett. B **671**, 10–14 (2009). arXiv:0707.0622

43. R. Trotta, F. Feroz, M. Hobson, L. Roszkowski, R. Ruiz de Austri, The impact of priors and observables on parameter inferences in the constrained MSSM. JHEP **12**, 24 (2008). arXiv:0809.3792

44. G.D. Martinez, J.S. Bullock, M. Kaplinghat, L.E. Strigari, R. Trotta, Indirect Dark Matter detection from Dwarf satellites: joint expectations from astrophysics and supersymmetry. JCAP **6**, 14 (2009). arXiv:0902.4715

45. L. Roszkowski, R. Ruiz de Austri, R. Trotta, Y.-L.S. Tsai, T.A. Varley, Global fits of the nonuniversal Higgs model. Phys. Rev. D **83**, 015014 (2011). arXiv:0903.1279

46. L. Roszkowski, R. Ruiz de Austri, R. Trotta, Efficient reconstruction of constrained MSSM parameters from LHC data: a case study. Phys. Rev. D **82**, 055003 (2010). arXiv:0907.0594

47. P. Scott, J. Conrad et al., Direct constraints on minimal supersymmetry from Fermi-LAT observations of the dwarf galaxy Segue 1. JCAP **1**, 31 (2010). arXiv:0909.3300

48. G. Bertone, D.G. Cerdeño, M. Fornasa, R. Ruiz de Austri, R. Trotta, Identification of dark matter particles with LHC and direct detection data. Phys. Rev. D **82**, 055008 (2010). arXiv:1005.4280

49. M. Bridges, K. Cranmer et al., A coverage study of CMSSM based on ATLAS sensitivity using fast neural networks techniques. JHEP **3**, 12 (2011). arXiv:1011.4306

50. C. Strege, G. Bertone et al., Updated global fits of the cMSSM including the latest LHC SUSY and Higgs searches and XENON100 data. JCAP **3**, 30 (2012). arXiv:1112.4192

51. G. Bertone, D. Cumberbatch, R. Ruiz de Austri, R. Trotta, Dark matter searches: the nightmare scenario. JCAP **1**, 4 (2012). arXiv:1107.5813

52. G. Bertone, D.G. Cerdeño et al., Complementarity of indirect and accelerator dark matter searches. Phys. Rev. D **85**, 055014 (2012). arXiv:1111.2607

53. P. Scott, C. Savage, J. Edsjö, the IceCube Collaboration, R. Abbasi et al., Use of event-level neutrino telescope data in global fits for theories of new physics. JCAP **11**, 57 (2012). arXiv:1207.0810

54. C. Strege, G. Bertone et al., Global fits of the cMSSM and NUHM including the LHC Higgs discovery and new XENON100 constraints. JCAP **4**, 13 (2013). arXiv:1212.2636

55. G. Bertone, D.G. Cerde no et al., Global fits of the cMSSM including the first LHC and XENON100 data. JCAP **1**, 15 (2012). arXiv:1107.1715

56. G. Bertone, F. Calore et al., Global analysis of the pMSSM in light of the Fermi GeV excess: prospects for the LHC Run-II and astroparticle experiments, arXiv:1507.07008

57. P. Bechtle, K. Desch, P. Wienemann, Fittino, a program for determining MSSM parameters from collider observables using an iterative method. Comput. Phys. Commun. **174**, 47–70 (2006). arXiv:hep-ph/0412012

58. P. Bechtle, K. Desch, M. Uhlenbrock, P. Wienemann, Constraining SUSY models with Fittino using measurements before, with and beyond the LHC. Eur. Phys. J. C **66**, 215–259 (2010). arXiv:0907.2589

59. P. Bechtle, T. Bringmann et al., Constrained supersymmetry after two years of LHC data: a global view with Fittino. JHEP **6**, 98 (2012). arXiv:1204.4199

60. O. Buchmueller, R. Cavanaugh et al., Predictions for supersymmetric particle masses using indirect experimental and cosmological constraints. JHEP **9**, 117 (2008). arXiv:0808.4128

61. O. Buchmueller, R. Cavanaugh et al., Likelihood functions for supersymmetric observables in frequentist analyses of the CMSSM and NUHM1. Eur. Phys. J. C **64**, 391–415 (2009). arXiv:0907.5568

62. O. Buchmueller, R. Cavanaugh et al., Frequentist analysis of the parameter space of minimal supergravity. Eur. Phys. J. C **71**, 1583 (2011). arXiv:1011.6118

63. O. Buchmueller, R. Cavanaugh et al., Implications of initial LHC searches for supersymmetry. Eur. Phys. J. C **71**, 1634 (2011). arXiv:1102.4585

64. O. Buchmueller, R. Cavanaugh et al., Supersymmetry and dark matter in light of LHC 2010 and XENON100 data. Eur. Phys. J. C **71**, 1722 (2011). arXiv:1106.2529

65. O. Buchmueller, R. Cavanaugh et al., Supersymmetry in light of 1/fb of LHC data. Eur. Phys. J. C **72**, 1878 (2012). arXiv:1110.3568

66. O. Buchmueller, R. Cavanaugh et al., Higgs and supersymmetry. Eur. Phys. J. C **72**, 2020 (2012). arXiv:1112.3564

67. O. Buchmueller, R. Cavanaugh et al., The CMSSM and NUHM1 in light of 7 TeV LHC, $B_s \to \mu^+\mu-$ and XENON100 data. Eur. Phys. J. C **72**, 2243 (2012). arXiv:1207.7315

68. O. Buchmueller et al., The CMSSM and NUHM1 after LHC run 1. Eur. Phys. J. C **74**, 2922 (2014). arXiv:1312.5250

69. O. Buchmueller et al., The NUHM2 after LHC run 1. Eur. Phys. J. C **74**, 3212 (2014). arXiv:1408.4060

70. E. Bagnaschi et al., Likelihood analysis of supersymmetric SU(5) GUTs. Eur. Phys. J. C **77**, 104 (2017). arXiv:1610.10084

71. E. Bagnaschi et al., Likelihood analysis of the minimal AMSB model. Eur. Phys. J. C **77**, 268 (2017). arXiv:1612.05210

72. B.C. Allanach, K. Cranmer, C.G. Lester, A.M. Weber, Natural priors. CMSSM fits and LHC weather forecasts. JHEP **08**, 023 (2007). arXiv:0705.0487

73. S.S. Abdussalam, B.C. Allanach, F. Quevedo, F. Feroz, M. Hobson, Fitting the phenomenological MSSM. Phys. Rev. D **81**, 095012 (2010). arXiv:0904.2548

74. S.S. Abdussalam, B.C. Allanach, M.J. Dolan, F. Feroz, M.P. Hobson, Selecting a model of supersymmetry breaking mediation. Phys. Rev. D **80**, 035017 (2009). arXiv:0906.0957

75. B.C. Allanach, Impact of CMS multi-jets and missing energy search on CMSSM fits. Phys. Rev. D **83**, 095019 (2011). arXiv:1102.3149

76. B.C. Allanach, T.J. Khoo, C.G. Lester, S.L. Williams, The impact of ATLAS zero-lepton, jets and missing momentum search on a CMSSM fit. JHEP **6**, 35 (2011). arXiv:1103.0969

77. A. Fowlie, A. Kalinowski, M. Kazana, L. Roszkowski, Y.L.S. Tsai, Bayesian implications of current LHC and XENON100 search limits for the constrained MSSM. Phys. Rev. D **85**, 075012 (2012). arXiv:1111.6098

78. L. Roszkowski, E.M. Sessolo, Y.-L.S. Tsai, Bayesian implications of current LHC supersymmetry and dark matter detection searches for the constrained MSSM. Phys. Rev. D **86**, 095005 (2012). arXiv:1202.1503

79. C. Balázs, A. Buckley, D. Carter, B. Farmer, M. White, Should we still believe in constrained supersymmetry? Eur. Phys. J. C **73**, 2563 (2013). arXiv:1205.1568

80. M.E. Cabrera, J.A. Casas, R. Ruiz de Austri, The health of SUSY after the Higgs discovery and the XENON100 data. JHEP **07**, 182 (2013). arXiv:1212.4821

81. A. Fowlie, K. Kowalska, L. Roszkowski, E.M. Sessolo, Y.-L.S. Tsai, Dark matter and collider signatures of the MSSM. Phys. Rev. D **88**, 055012 (2013). arXiv:1306.1567

82. S. Henrot-Versillé, R. Lafaye et al., Constraining supersymmetry using the relic density and the Higgs boson. Phys. Rev. D **89**, 055017 (2014). arXiv:1309.6958

83. D. Kim, P. Athron, C. Balázs, B. Farmer, E. Hutchison, Bayesian naturalness of the CMSSM and CNMSSM. Phys. Rev. D **90**, 055008 (2014). arXiv:1312.4150

84. A. Fowlie, M. Raidal, Prospects for constrained supersymmetry at $\sqrt{s} = 33$ TeV and $\sqrt{s} = 100$ TeV proton-proton super-colliders. Eur. Phys. J. C **74**, 2948 (2014). arXiv:1402.5419

85. L. Roszkowski, E.M. Sessolo, A.J. Williams, What next for the CMSSM and the NUHM: improved prospects for superpartner and dark matter detection. JHEP **08**, 067 (2014). arXiv:1405.4289

86. K. Kowalska, L. Roszkowski, E.M. Sessolo, A.J. Williams, GUT-inspired SUSY and the muon $g - 2$ anomaly: prospects for LHC 14 TeV. JHEP **06**, 020 (2015). arXiv:1503.08219

87. M.E. Cabrera, J.A. Casas, A. Delgado, S. Robles, R. Ruiz de Austri, Naturalness of MSSM dark matter. JHEP **08**, 058 (2016). arXiv:1604.02102

88. C. Han, K.-I. Hikasa, L. Wu, J. M. Yang, Y. Zhang, Status of CMSSM in light of current LHC run-2 and LUX data. arXiv:1612.02296

89. C. Strege, G. Bertone et al., Profile likelihood maps of a 15-dimensional MSSM. JHEP **9**, 81 (2014). arXiv:1405.0622

90. P. Bechtle, J.E. Camargo-Molina et al., Killing the cMSSM softly. Eur. Phys. J. C **76**, 96 (2016). arXiv:1508.05951

91. M.E. Cabrera-Catalan, S. Ando, C. Weniger, F. Zandanel, Indirect and direct detection prospect for TeV dark matter in the nine parameter MSSM. Phys. Rev. D **92**, 035018 (2015). arXiv:1503.00599

92. K.J. de Vries, E.A. Bagnaschi et al., The pMSSM10 after LHC run 1. Eur. Phys. J. C **75**, 422 (2015). arXiv:1504.03260

93. E.A. Bagnaschi, O. Buchmueller et al., Supersymmetric dark matter after LHC run 1. Eur. Phys. J. C **75**, 500 (2015). arXiv:1508.01173

94. C. Balázs, D. Carter, Discovery potential of the next-to-minimal supergravity-motivated model. Phys. Rev. D **78**, 055001 (2008). arXiv:0808.0770

95. D.E. Lopez-Fogliani, L. Roszkowski, R.R. de Austri, T.A. Varley, A Bayesian analysis of the constrained NMSSM. Phys. Rev. D **80**, 095013 (2009). arXiv:0906.4911

96. K. Kowalska, S. Munir et al., Constrained next-to-minimal supersymmetric standard model with a 126 GeV Higgs boson: a global analysis. Phys. Rev. D **87**, 115010 (2013). arXiv:1211.1693

97. A. Fowlie, Is the CNMSSM more credible than the CMSSM? Eur. Phys. J. C **74**, 3105 (2014). arXiv:1407.7534

98. G. Bertone, K. Kong, R.R. de Austri, R. Trotta, Global fits of the minimal universal extra dimensions scenario. Phys. Rev. D **83**, 036008 (2011). arXiv:1010.2023

99. K. Cheung, Y.-L.S. Tsai, P.-Y. Tseng, T.-C. Yuan, A. Zee, Global study of the simplest scalar phantom dark matter model. JCAP **1210**, 042 (2012). arXiv:1207.4930

100. A. Arhrib, Y.-L.S. Tsai, Q. Yuan, T.-C. Yuan, An updated analysis of inert Higgs doublet model in light of the recent results from LUX, PLANCK, AMS-02 and LHC. JCAP **1406**, 030 (2014). arXiv:1310.0358

101. S. Matsumoto, S. Mukhopadhyay, Y.-L.S. Tsai, Singlet Majorana fermion dark matter: a comprehensive analysis in effective field theory. JHEP **10**, 155 (2014). arXiv:1407.1859

102. D. Chowdhury, O. Eberhardt, Global fits of the two-loop renormalized two-Higgs-doublet model with soft Z_2 breaking. JHEP **11**, 52 (2015). arXiv:1503.08216

103. S. Liem, G. Bertone et al., Effective field theory of dark matter: a global analysis. JHEP **9**, 77 (2016). arXiv:1603.05994

104. X. Huang, Y.-L .S. Tsai, Q. Yuan, LikeDM: likelihood calculator of dark matter detection. Comput. Phys. Commun. **213**, 252–263 (2017). arXiv:1603.07119

105. S. Banerjee, S. Matsumoto, K. Mukaida, Y.-L.S. Tsai, WIMP dark matter in a well-tempered regime: a case study on singlet-doublets Fermionic WIMP. JHEP **11**, 070 (2016). arXiv:1603.07387

106. S. Matsumoto, S. Mukhopadhyay, Y.-L.S. Tsai, Effective theory of WIMP dark matter supplemented by simplified models: singlet-like Majorana fermion case. Phys. Rev. D **94**, 065034 (2016). arXiv:1604.02230

107. A. Cuoco, B. Eiteneuer, J. Heisig, M. Krämer, A global fit of the γ-ray galactic center excess within the scalar singlet Higgs portal model. JCAP **6**, 050 (2016). arXiv:1603.08228

108. GAMBIT Collider Workgroup: C. Balázs, A. Buckley et al., ColliderBit: a GAMBIT module for the calculation of high-energy collider observables and likelihoods. arXiv:1705.07919

109. GAMBIT Flavour Workgroup, F. U. Bernlochner, M. Chrzaszcz et al., FlavBit: a GAMBIT module for computing flavour observables and likelihoods. arXiv:1705.07933

110. GAMBIT Dark Matter Workgroup, T. Bringmann, J. Conrad et al., DarkBit: a GAMBIT module for computing dark matter observables and likelihoods. arXiv:1705.07920

111. GAMBIT Models Workgroup, P. Athron, C. Balázs et al., SpecBit, DecayBit and PrecisionBit: GAMBIT modules for computing mass spectra, particle decay rates and precision observables. arXiv:1705.07936

112. GAMBIT Scanner Workgroup, G. D. Martinez, J. McKay et al., Comparison of statistical sampling methods with ScannerBit, the GAMBIT scanning module. arXiv:1705.07959

113. R. Ruiz de Austri, R. Trotta, F. Feroz, SuperBayeS. http://www.superbayes.org

114. Y. Akrami, P. Scott, J. Edsjö, J. Conrad, L. Bergström, A profile likelihood analysis of the constrained MSSM with genetic algorithms. JHEP **4**, 57 (2010). arXiv:0910.3950

115. F. Feroz, K. Cranmer, M. Hobson, R. Ruiz de Austri, R. Trotta, Challenges of profile likelihood evaluation in multi-dimensional SUSY scans. JHEP **6**, 42 (2011). arXiv:1101.3296

116. Y. Akrami, C. Savage, P. Scott, J. Conrad, J. Edsjö, Statistical coverage for supersymmetric parameter estimation: a case study with direct detection of dark matter. JCAP **7**, 2 (2011). arXiv:1011.4297

117. C. Strege, R. Trotta, G. Bertone, A.H.G. Peter, P. Scott, Fundamental statistical limitations of future dark matter direct detection experiments. Phys. Rev. D **86**, 023507 (2012). arXiv:1201.3631

118. C.F. Berger, J.S. Gainer, J.A.L. Hewett, T.G. Rizzo, Supersymmetry without prejudice. JHEP **2**, 23 (2009). arXiv:0812.0980

119. GAMBIT Collaboration, P. Athron, C. Balázs et al., Global fits of GUT-scale SUSY models with GAMBIT. arXiv:1705.07935

120. GAMBIT Collaboration, P. Athron, C. Balázs et al., A global fit of the MSSM with GAMBIT. arXiv:1705.07917

121. GAMBIT Collaboration, P. Athron, C. Balázs et al., Status of the scalar singlet dark matter model. arXiv:1705.07931

122. M. Cacciari, G.P. Salam, G. Soyez, FastJet user manual. Eur. Phys. J. C **72**, 1896 (2012). arXiv:1111.6097

123. P. Athron, J.-H. Park, D. Stöckinger, A. Voigt, FlexibleSUSY—a spectrum generator generator for supersymmetric models. Comput. Phys. Commun. **190**, 139–172 (2015). arXiv:1406.2319

124. B.C. Allanach, SOFTSUSY: a program for calculating supersymmetric spectra. Comput. Phys. Commun. **143**, 305–331 (2002). arXiv:hep-ph/0104145

125. T. Sjostrand, S. Ask et al., An introduction to PYTHIA 8.2. Comput. Phys. Commun. **191**, 159–177 (2015). arXiv:1410.3012

126. S. Ovyn, X. Rouby, V. Lemaitre, DELPHES, a framework for fast simulation of a generic collider experiment. arXiv:0903.2225

127. J. de Favereau et al., DELPHES 3, a modular framework for fast simulation of a generic collider experiment. JHEP **1402**, 057 (2014). arXiv:1307.6346

128. P. Bechtle, O. Brein, S. Heinemeyer, G. Weiglein, K.E. Williams, HiggsBounds: confronting arbitrary Higgs sectors with exclusion bounds from LEP and the tevatron. Comput. Phys. Commun. **181**, 138–167 (2010). arXiv:0811.4169

129. P. Bechtle, O. Brein, S. Heinemeyer, G. Weiglein, K.E. Williams, HiggsBounds 2.0.0: confronting neutral and charged Higgs sector predictions with exclusion bounds from LEP and the tevatron. Comput. Phys. Commun. **182**, 2605–2631 (2011). arXiv:1102.1898

130. P. Bechtle, O. Brein et al., $HiggsBounds - 4$: improved tests of extended Higgs sectors against exclusion bounds from LEP, the tevatron and the LHC. Eur. Phys. J. C **74**, 2693 (2014). arXiv:1311.0055

131. P. Bechtle, S. Heinemeyer, O. Stål, T. Stefaniak, G. Weiglein, HiggsSignals: confronting arbitrary Higgs sectors with measurements at the tevatron and the LHC. Eur. Phys. J. C **74**, 2711 (2014). arXiv:1305.1933

132. J. Alwall, M. Herquet, F. Maltoni, O. Mattelaer, T. Stelzer, MadGraph 5: going beyond. JHEP **06**, 128 (2011). arXiv:1106.0522

133. J. Alwall, R. Frederix et al., The automated computation of tree-level and next-to-leading order differential cross sections, and their matching to parton shower simulations. JHEP **07**, 079 (2014). arXiv:1405.0301

134. F. Mahmoudi, SuperIso: a program for calculating the isospin asymmetry of $B \to K^* \gamma$ in the MSSM. Comput. Phys. Commun. **178**, 745 (2008). arXiv:0710.2067

135. F. Mahmoudi, SuperIso v2.3: a program for calculating flavor physics observables in supersymmetry. Comput. Phys. Commun. **180**, 1579 (2009). arXiv:0808.3144

136. F. Mahmoudi, SuperIso v3.0, flavor physics observables calculations: extension to NMSSM. Comput. Phys. Commun. **180**, 1718 (2009)

137. H. Bahl, S. Heinemeyer, W. Hollik, G. Weiglein, Reconciling EFT and hybrid calculations of the light MSSM Higgs-boson mass. arXiv:1706.00346

138. 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

139. 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

140. 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

141. 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

142. 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

143. 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

144. F. Mahmoudi, S. Neshatpour, J. Virto, $B \to K^* \mu^+ \mu^-$ optimised observables in the MSSM. Eur. Phys. J. C **74**, 2927 (2014). arXiv:1401.2145

145. W. Altmannshofer, D.M. Straub, New physics in $B \to K^* \mu\mu$? Eur. Phys. J. C **73**, 2646 (2013). arXiv:1308.1501

146. S. Descotes-Genon, L. Hofer, J. Matias, J. Virto, Global analysis of $b \to s\ell\ell$ anomalies. JHEP **06**, 092 (2016). arXiv:1510.04239

147. P. Gondolo, J. Edsjö et al., DarkSUSY: computing supersymmetric dark matter properties numerically. JCAP **7**, 8 (2004). arXiv:astro-ph/0406204

148. G. Bélanger, F. Boudjema, A. Pukhov, A. Semenov, MicrOMEGAs: a program for calculating the relic density in the MSSM. Comput. Phys. Commun. **149**, 103–120 (2002). arXiv:hep-ph/0112278

149. G. Bélanger, F. Boudjema, A. Pukhov, A. Semenov, micrOMEGAs: version 1.3. Comput. Phys. Commun. **174**, 577–604 (2006). arXiv:hep-ph/0405253

150. G. Bélanger, F. Boudjema, A. Pukhov, A. Semenov, MicrOMEGAs 2.0: a program to calculate the relic density of dark matter in a generic model. Comput. Phys. Commun. **176**, 367–382 (2007). arXiv:hep-ph/0607059

151. G. Bélanger, F. Boudjema et al., Indirect search for dark matter with micrOMEGAs2.4. Comput. Phys. Commun. **182**, 842–856 (2011). arXiv:1004.1092

152. G. Bélanger, F. Boudjema, A. Pukhov, A. Semenov, micrOMEGAs 3: a program for calculating dark matter observables. Comput. Phys. Commun. **185**, 960–985 (2014). arXiv:1305.0237

153. G. Bélanger, F. Boudjema, A. Pukhov, A. Semenov, micrOMEGAs4.1: two dark matter candidates. Comput. Phys. Commun. **192**, 322–329 (2015). arXiv:1407.6129

154. 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

155. 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

156. 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

157. 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

158. 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

159. 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

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

161. 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

162. 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

163. 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

164. 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

165. Message Passing Forum, *MPI: A Message-Passing Interface Standard* (University of Tennessee, Knoxville, 1994)

166. L. Dagum, R. Menon, OpenMP: an industry standard API for shared-memory programming. IEEE Comput. Sci. Eng. **5**, 46–55 (1998)

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

168. H. Baer, X. Tata, *Weak Scale Supersymmetry* (Cambridge University Press, Cambridge, 2006)

169. H. Silverwood, P. Scott et al., Sensitivity of IceCube-DeepCore to neutralino dark matter in the MSSM-25. JCAP **3**, 27 (2013). arXiv:1210.0844

170. R.C. Cotta, A. Drlica-Wagner et al., Constraints on the pMSSM from LAT observations of dwarf spheroidal galaxies. JCAP **4**, 16 (2012). arXiv:1111.2604

171. J.A. Conley, J.S. Gainer, J.L. Hewett, M.P. Le, T.G. Rizzo, Supersymmetry without prejudice at the LHC. Eur. Phys. J. C **71**, 1697 (2011). arXiv:1009.2539

172. A. Arbey, M. Battaglia, F. Mahmoudi, Implications of LHC searches on SUSY particle spectra. The pMSSM parameter space with neutralino dark matter. Eur. Phys. J. C **72**, 1847 (2012). arXiv:1110.3726

173. L. Bergström, P. Gondolo, Limits on direct detection of neutralino dark matter from $b \to s\gamma$ decays. Astropart. Phys. **5**, 263–278 (1996). arXiv:hep-ph/9510252

174. M. Berg, J. Edsjö, P. Gondolo, E. Lundström, S. Sjörs, Neutralino dark matter in BMSSM effective theory. JCAP **8**, 35 (2009). arXiv:0906.0583

175. E. Dudas, Y. Mambrini, A. Mustafayev, K.A. Olive, Relating the CMSSM and SUGRA models with GUT scale and super-GUT scale supersymmetry breaking. Eur. Phys. J. C **72**, 2138 (2012). arXiv:1205.5988. [Erratum: Eur. Phys. J. C **73**, 2430 (2013)]

176. Y. Akrami, C. Savage, P. Scott, J. Conrad, J. Edsjö, How well will ton-scale dark matter direct detection experiments constrain minimal supersymmetry? JCAP **1104**, 012 (2011). arXiv:1011.4318

177. R. Schoenrich, J. Binney, W. Dehnen, Local kinematics and the local standard of rest. MNRAS **403**, 1829 (2010). arXiv:0912.3693

178. J.R. Ellis, K.A. Olive, C. Savage, Hadronic uncertainties in the elastic scattering of supersymmetric dark matter. Phys. Rev. D **77**, 065026 (2008). arXiv:0801.3656

179. R. D. Young, Strange quark content of the nucleon and dark matter searches. PoS **LATTICE2012**, 014 (2012). arXiv:1301.1765

180. 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

181. P. Scott, Pippi—painless parsing, post-processing and plotting of posterior and likelihood samples. Eur. Phys. J. Plus **127**, 138 (2012). arXiv:1206.2245

182. F. Feroz, M.P. Hobson, M. Bridges, MULTINEST: an efficient and robust Bayesian inference tool for cosmology and particle physics. MNRAS **398**, 1601–1614 (2009). arXiv:0809.3437

183. A. Putze, L. Derome, The Grenoble Analysis Toolkit (GreAT)—a statistical analysis framework. Phys. Dark Univ. **5**, 29–34 (2014)

184. A. Lewis, Efficient sampling of fast and slow cosmological parameters. Phys. Rev. D **87**, 103529 (2013). arXiv:1304.4473

185. A. Lewis, S. Bridle, Cosmological parameters from CMB and other data: a Monte Carlo approach. Phys. Rev. D **66**, 103511 (2002). arXiv:astro-ph/0205436