Eur. Phys. J. C (2017) 77:761
DOI 10.1140/epjc/s10052-017-5274-y

THE EUROPEAN PHYSICAL JOURNAL C

Special Article - Tools for Experiment and Theory

Comparison of statistical sampling methods with ScannerBit, the GAMBIT scanning module

The GAMBIT Scanner Workgroup: Gregory D. Martinez[1,a], James McKay[2,b], Ben Farmer[3,4,c], Pat Scott[2,d], Elinore Roebber[5], Antje Putze[6], Jan Conrad[3,4]

[1] Physics and Astronomy Department, University of California, Los Angeles, CA 90095, USA
[2] Department of Physics, Blackett Laboratory, Imperial College London, Prince Consort Road, London SW7 2AZ, UK
[3] Oskar Klein Centre for Cosmoparticle Physics, AlbaNova University Centre, 10691 Stockholm, Sweden
[4] Department of Physics, Stockholm University, 10691 Stockholm, Sweden
[5] Department of Physics, McGill University, 3600 rue University, Montreal, QC H3A 2T8, Canada
[6] LAPTh, Université de Savoie, CNRS, 9 chemin de Bellevue B.P.110, 74941 Annecy-le-Vieux, France

Received: 14 March 2017 / Accepted: 2 October 2017
© The Author(s) 2017. This article is an open access publication

Abstract We introduce ScannerBit, the statistics and sampling module of the public, open-source global fitting framework GAMBIT. ScannerBit provides a standardised interface to different sampling algorithms, enabling the use and comparison of multiple computational methods for inferring profile likelihoods, Bayesian posteriors, and other statistical quantities. The current version offers random, grid, raster, nested sampling, differential evolution, Markov Chain Monte Carlo (MCMC) and ensemble Monte Carlo samplers. We also announce the release of a new standalone differential evolution sampler, Diver, and describe its design, usage and interface to ScannerBit. We subject Diver and three other samplers (the nested sampler MultiNest, the MCMC GreAT, and the native ScannerBit implementation of the ensemble Monte Carlo algorithm T-Walk) to a battery of statistical tests. For this we use a realistic physical likelihood function, based on the scalar singlet model of dark matter. We examine the performance of each sampler as a function of its adjustable settings, and the dimensionality of the sampling problem. We evaluate performance on four metrics: optimality of the best fit found, completeness in exploring the best-fit region, number of likelihood evaluations, and total runtime. For Bayesian posterior estimation at high resolution, T-Walk provides the most accurate and timely mapping of the full parameter space. For profile likelihood analysis in less than about ten dimensions, we find that Diver and MultiNest score similarly in terms of best fit and speed, outperforming GreAT and T-Walk; in ten or more dimensions, Diver substantially outperforms the other three samplers on all metrics.

Contents

1 Introduction . . . .
2 Package description . . . .
2.1 ScannerBit plugins . . . .
3 Statistics and scanning . . . .
3.1 Priors and sampling distributions . . . .
3.1.1 Built-in one-dimensional priors . . . .
3.1.2 Built-in multi-dimensional priors . . . .
3.1.3 Additional built-in priors . . . .
3.2 Plugins . . . .
4 Setup and input file options . . . .
4.1 Input file `Parameters` section . . . .
4.2 Input file `Priors` section . . . .
4.3 Input file `Scanner` section . . . .
4.4 ScannerBit standalone executable . . . .
5 Simple scanners . . . .
5.1 The random sampler . . . .
5.2 The grid and square_grid scanners . . . .
5.3 The raster scanner . . . .
5.4 The toy_mcmc scanner . . . .
6 The postprocessor . . . .
7 Markov Chain Monte Carlo . . . .
7.1 The GreAT software . . . .
7.2 GreAT–ScannerBit interface . . . .
8 Ensemble MCMC . . . .
8.1 T-Walk . . . .
9 Nested sampling . . . .

[a] e-mail: gmartine@astro.ucla.edu
[b] e-mail: j.mckay14@imperial.ac.uk
[c] e-mail: benjamin.farmer@fysik.su.se
[d] e-mail: p.scott@imperial.ac.uk

Published online: 13 November 2017

10 Differential evolution
10.1 Algorithmic details
10.1.1 Mutation
10.1.2 Crossover
10.1.3 Selection
10.1.4 Advanced mutation and crossover strategies
10.1.5 Self-adaptive differential evolution
10.2 The Diver package
10.2.1 Design and invocation
10.2.2 Adaptive differential evolution: jDE and λjDE
10.2.3 Discrete parameters and parameter-space partitioning
10.2.4 Population diversity and duplicate individuals
10.2.5 Approximate posterior and evidence estimates
10.2.6 ScannerBit interface
11 Scanner performance comparisons
11.1 MultiNest
11.2 Diver
11.3 T-Walk
11.4 GreAT
11.5 The effect of dimensionality on performance
11.6 Scanning efficiency
11.7 Posterior sampling
11.8 Discussion
12 Conclusions
Appendix A: Sources, options and outputs of the Diver package
A.1: Sources
A.2: Run options
A.3: Output formats
Appendix B: Scanner options and outputs
B.1: Postprocessor
B.2: GreAT
B.3: T-Walk
B.4: MultiNest
B.5: Diver
Appendix C: Custom priors
Appendix D: Plugin Declaration and Interface
D.1: Plugin declaration
D.2: Interface to input file
D.3: Interface to prior object
D.4: Interface to GAMBIT printer system
D.5: Scanner plugins
D.5.1: Scanner plugin example
D.6: Objective plugins
D.6.1: Objective plugin example
Appendix E: Scanner comparisons in a two-dimensional parameter space
E.1: MultiNest & Diver
E.2: T-Walk
E.3: GreAT
E.4: Summary
Appendix F: YAML input file example
Appendix G: Glossary
References

1 Introduction

Science has entered an era of increasing computational complexity. Large data sets and burgeoning model complexity have necessitated the development of increasingly sophisticated and efficient analysis techniques. As datasets and theories in particle physics and cosmology have become more computationally expensive to work with, the problem of efficiently and comprehensively sampling model parameter spaces has become steadily more challenging. Simple random parameter sampling (e.g. [1,2]) has gradually proven more inadequate as time goes on, as it typically leads to incomplete and biased inferences when applied to all but the simplest problems.

Workers in various fields have employed increasingly advanced numerical and statistical methods to deal with this challenge. Bayesian numerical techniques such as Markov Chain Monte Carlos (MCMCs) became particularly popular in cosmology, because of their theoretical near-linear scalability with parameter dimensionality. Cosmic Microwave Background (CMB) analyses were amongst the first such applications of MCMCs [3], with later improvements and optimisations brought about through the use of adaptive techniques and robust convergence criteria [4–6]. MCMCs also proved popular in particle physics, for the exploration of moderately complex supersymmetric model parameter spaces [7–11]. Nested sampling [12] gradually displaced MCMCs in many such applications [13–16], owing to its efficiency for mapping posterior distributions and calculating the Bayesian evidence, especially when dealing with multimodal likelihoods [17].

Because the likelihood functions involved are computationally expensive (see e.g. [18,19]), fully frequentist Neyman constructions are typically not possible. A popular alternative to Bayesian inference is to examine the prior-independent profile likelihood. However, Bayesian methods such as MCMCs and nested sampling are not necessarily optimal sampling strategies in this case [20]. Estimating the Bayesian posterior requires integrating the likelihood in various directions of the parameter space, whereas the profile likelihood relies instead on maximising it in those directions. From the perspective of numerical analysis, to a first approximation Bayesian sampling is an integration problem, whereas profile likelihood estimation is an optimisation problem. It

is therefore unsurprising that modern multi-modal optimisation strategies such as genetic algorithms and differential evolution have proven more efficient than Bayesian methods in some applications of the profile likelihood [20,21].

This picture is further complicated by additional requirements not present in traditional optimisation problems. To be able to infer reliable confidence intervals on parameters, the likelihood function must be sampled sufficiently well around the maximum to allow isolikelihood contours to be inferred. Unfortunately, determination of the global best-fit point does not necessarily guarantee that this will be the case. In this respect, some Bayesian methods can in fact be more efficient than optimisers, even if they are less efficient at finding the global maximum [22]. Another issue is the degree to which the resulting confidence intervals achieved in the profile likelihood analysis have the expected statistical coverage properties [23–26]; this can be strongly influenced by the choice of scanning algorithm.

In this paper we provide a detailed manual for ScannerBit, a package designed to provide a common interface to a range of different sampling algorithms, so that the performance of the different algorithms can be easily compared, and the most appropriate algorithm (or combination thereof) chosen for the problem at hand. We also carry out some such comparisons of sampling algorithms, and provide recommended settings for different samplers.

ScannerBit is designed to be modular and expandable, allowing it to access a multitude of different samplers in a plug and play fashion. As the GAMBIT project grows, we will continually add scanners to the ScannerBit suite. Users can also easily implement various scanners to meet their personal needs. ScannerBit initially ships with four production-quality scanners: an adaptive MCMC (GreAT), an ensemble MCMC (T-Walk), a nested sampler (MultiNest) and a differential evolution sampler (Diver). GreAT [27] and MultiNest [17] are existing external packages. Diver is a new external package that we describe for the first time here. T-Walk is implemented natively in ScannerBit. The ScannerBit package also contains a postprocessor and a series of simple scanners, including random, grid and list-based samplers and a more basic toy MCMC (for tutorial purposes).

All the scanners initially accessible from ScannerBit are designed for the calculation of profile likelihoods or Bayesian posteriors, such that they select optimal parameter combinations for which to perform likelihood calculations. These samplers therefore require the likelihood to be explicitly calculable for any parameter combination, either by parametrisation or numerical approximation. The design of ScannerBit is not limited to this operation mode, however, and can easily support methods that do not require explicit calculation of a likelihood, such as Approximate Bayesian Computation [28].

ScannerBit can either be used within its parent code GAMBIT [29], or as a standalone package, or simply interfaced directly to an external likelihood function.

We begin by describing the ScannerBit package in Sect. 2, before giving the implementation details and the underlying statistical methods that we employ in Sect. 3. The user interface is covered in Sect. 4, and the simple scanners in Sect. 5. Sections 6–10 respectively describe the postprocessor, MCMC, ensemble MCMC, nested sampler and differential evolution samplers. In Sect. 11 we perform a detailed comparison of the different algorithms implemented in ScannerBit, and their available parameters and options. We summarise in Sect. 12, then provide an extensive set of appendices. These cover the sources, options and outputs of our differential evolution sampler Diver (Appendix A), ScannerBit options and outputs for all five major scanners (Appendix B), examples of how to implement new priors (Appendix C), examples of adding new scanners and objective functions (Appendix D), some supplementary comparisons of scanner performance (Appendix E), a minimal example input file (Appendix F), and a glossary of the most commonly-used GAMBIT terms (Appendix G).

More details on GAMBIT itself can be found in Ref. [29], on its various physics modules in Refs. [19,30–32], and on first physics results in Refs. [33–35].

2 Package description

To efficiently sample an n-dimensional parameter space, ScannerBit works by separating the sampling problem into three distinct steps:

1. Choosing n values in the interval between 0 and 1. Taken together, these values constitute a 'point' in the n-dimensional 'unit hypercube'.
2. Transforming the point in the unit hypercube into a point in the physical n-dimensional parameter space.
3. Passing the values of the physical parameters to a user-specified function, which may compute a number of things from the parameter values, including theoretical predictions of different observable quantities and corresponding likelihoods, based on e.g. comparison with experimental data.

The steps then repeat until some convergence criterion is satisfied, with the results of Step 3 used to help choose the next point in the unit hypercube in the subsequent iteration of Step 1.

The results of Step 3 are output in each iteration, in a format of the user's choice. Any output format supported by the GAMBIT printer system can be chosen, including ASCII and HDF5 database files. The GAMBIT printers are described in detail in Sect. 9 of Ref. [29], and ScannerBit's interface

to them is described in Sect. D.4. Both ASCII and HDF5 outputs can be parsed and plotted as profile likelihoods with pippi [36], which can be installed automatically from within GAMBIT (or ScannerBit) by typing `make get-pippi` from within the build directory. As the printer output is handled in Step 3, independent of the sampling algorithms responsible for Step 1, every parameter set tested is printed, along with every quantity derived from this set, regardless of whether the scanner accepts the tested point or not. This provides the maximum information possible for profile likelihood and post-processing analyses. The GAMBIT printers can also be sent additional supplementary data computed by the samplers themselves, such as the posterior weights needed for plotting posterior probability densities with pippi.

2.1 ScannerBit plugins

ScannerBit is designed to be completely modular and expandable. It achieves this via a plugin interface, which allows various scanners and likelihood functions to be connected at will. Plugins are either **scanner plugins**, which each contain code implementing a single sampling algorithm, or objective plugins, also known as **test function plugins**, which contain specific objective functions to be scanned (such as simple test functions and likelihoods). Scanner plugins are responsible for efficiently navigating the unit cube in Step 1, whereas objective plugins provide the user-specified function in Step 3. Each plugin is compiled into an independent library with a common interface to ScannerBit, so that at runtime it can be passed necessary information like the dimensionality of the space being scanned and the user's preferred method of outputting the results.

The transformation that must be applied in Step 2 constitutes a sampling prior. This is relevant both for Bayesian analyses, where the final posterior probability of the model parameters is directly proportional to the prior, and for profile likelihood analyses, where the sampling prior can have an impact on how efficiently the likelihood function can be sampled. ScannerBit implements priors as transformations of the uniform probability distribution, as it instructs all scanner plugins to carry out Step 1 by sampling the unit hypercube using a uniform sampling prior. ScannerBit transforms the samples generated from the unit hypercube into actual model-space parameter values by requiring the user to select a prior transformation to apply to each parameter. This allows scanner plugins to operate completely independently of priors. Sampler implementations are kept entirely independent of prior implementations, allowing any scanner to be used with any prior.[1] Priors can be added to ScannerBit in a similarly modular way to scanner and test function plugins (see Sect. 3.1).

ScannerBit grants scanner plugins access to specific functions necessary for them to perform their sampling task. At the simplest level, the only such functions are the prior transformation of Step 2, and a log-likelihood function for Step 3, allowing the likelihood to be evaluated for any given point in the hypercube. The function(s) provided to a scanner plugin at runtime are selected by assigning **purposes** (such as "`LogLike`") to different objective plugins or results provided by GAMBIT, and then telling each scanner which purpose(s) corresponding to the inputs it should collect. The purposes are specified in the input file for a ScannerBit run, which should be written in YAML format.[2] All ScannerBit objective functions tagged for a common purpose are combined into a single function, and provided to the scanner as a function pointer. In a regular GAMBIT scan, this is the total log-likelihood function provided by the **likelihood container**, which combines GAMBIT functions tagged with a common purpose, according to the specific function **capabilities** requested by the user in their input YAML file.

Generically, objective plugins take model parameter values as inputs, and return some quantity useful to ScannerBit for performing a scan. Each objective can be individually assigned a purpose to enable its output to be assigned appropriately in a scanner plugin. The canonical example of an objective plugin is the merit function to be used in a given scan, allowing ScannerBit to determine which parameter combinations are better than others, and to make informed choices about which combinations to sample next. This function might be a complicated likelihood (as in the case of the GAMBIT **likelihood container**), or just a simple test function for evaluating the performance of a new scanner. A more advanced example of an objective plugin would be one that provides the derivative of a merit function, for use with e.g. optimisers that use derivatives to accelerate their searches. Whilst each objective plugin is automatically given access to the prior chosen for a given scan, objective plugins can in fact *also* be employed to provide the underlying transformation function used in a prior (although this method is not mandatory for defining a new prior – see Sect. 3.1).

Each plugin's source code is placed in its own subdirectory within `ScannerBit/src/`*plugin_kind*, where *plugin_kind* is either `scanners` or `objectives`. The plugin headers reside in their own subdirectory within `ScannerBit/headers/gambit/ScannerBit/`*plugin_kind*. Each plugin's compilation and linkage is handled by the ScannerBit CMake script.

[1] Although scanning the unit hypercube is the default, ScannerBit does also permit special scanners developed for specific models to choose to bypass the prior transformation entirely, in order to work directly with model parameter values. Users are advised to avoid this unless strictly necessary though, as the resulting scanner is neither usable with other models nor other priors.

[2] http://www.yaml.org.

3 Statistics and scanning

To run a parameter scan in GAMBIT, the user writes an input YAML file specifying that they want to analyse a particular model. They indicate the parameter ranges and priors over which GAMBIT should sample that model, how that sampling should be done, and what quantities should be computed for each parameter combination. GAMBIT activates the model in its model database, along with all other models that the model in question is a subspace of. The dependency resolver uses the activated model hierarchy and the list of the user's requested quantities to activate and connect various module functions into a dependency graph (see Ref. [29]). ScannerBit is then responsible for determining which parameter combinations to run through the dependency graph.

Every quantity requested for calculation in a scan must be assigned a **purpose** in the input YAML file, using the eponymous option `purpose`. This can be set to `Test` or `Observable`, to flag that the quantity must be computed and output for each sample. To include the quantity in the function that actually drives a sampler, the user must match the `purpose` of the quantity to whatever `purpose` he or she instructs the sampler to seek out in order to define its objective function. Once dependency resolution has been completed, GAMBIT constructs a **likelihood container**, which consists of the dependency tree of all module functions assigned the purpose sought by the sampler. This container essentially packages the results of the different module functions into a single function that can be called by any sampling algorithm.

Conventionally, GAMBIT example YAML files assign `purpose:LogLike` to any quantity that should enter the fit as a likelihood component, and expects such functions to return the natural logarithm of the likelihood $\log \mathcal{L}$. By simply summing their return values, the likelihood container combines the results of all log-likelihood functions and returns the result to ScannerBit as the total log-likelihood. At present, the sampling algorithms callable by ScannerBit allow only a single `purpose` to dictate the behaviour of a scan, although future scanners are anticipated to make use of two or more distinct purposes in a single scan (as in e.g. in multi-objective optimisation).

3.1 Priors and sampling distributions

Most samplers are driven by ScannerBit to draw from the unit interval [0, 1]. The sampled values are then converted to real physical parameters internally, using whatever prior the user has chosen when launching the scan. In the simplest cases, this occurs by applying the transformation method, where samples from the unit interval are converted to samples from the desired sampling distribution (i.e. prior), by applying the inverse of the cumulative distribution function (CDF) of the desired distribution. Here, a uniform random deviate x is transformed into a random deviate y sampled from a target distribution D with cumulative distribution function $F(y)$, by computing

$$y = F^{-1}(x). \tag{1}$$

Take as an example the case where a user requests a flat 'prior' over the range $[a, b]$ for some parameter. ScannerBit expects the underlying sampler to provide a number x in the interval [0, 1], and then applies the transformation

$$y = F^{-1}(x) = (b - a)x + a, \tag{2}$$

in order to obtain a sample in the range $[a, b]$. Here $F^{-1}(x)$ is the inverse of

$$\begin{aligned} F(y) &\equiv \int_a^y P(x)\,dx \\ &= \int_a^y \frac{dx}{b-a} \\ &= \frac{y-a}{b-a}, \end{aligned} \tag{3}$$

which is the CDF of $P(x) \equiv 1/(b - a)$, the uniform distribution over $[a, b]$. Thus, although the underlying sampler chooses uniform random numbers for x from the interval [0, 1], the final 'physical' parameter y will be sampled uniformly from the interval $[a, b]$. Similarly, if the user requests a 'Gaussian' prior (with mean μ and standard deviation σ) for parameter y, then ScannerBit will apply the transformation

$$y = \mu + \sigma\sqrt{2}\ \mathrm{erf}^{-1}\,(2x - 1)\,, \tag{4}$$

so that uniform samples from the unit interval are transformed into samples from the normal distribution $\mathcal{N}(\mu, \sigma)$.

It is important to note that the actual sampling distribution of a scan only follows these transformed distributions in the special case where the underlying unit-interval sampling is actually uniform. This corresponds to the case of a purely random sampling algorithm (implemented as the random sampler in ScannerBit; see Sect. 5.1).

If the underlying sampling is driven, for example, by a Metropolis-Hastings algorithm, or an evolutionary sampler, then the final samples will of course not be drawn directly from the user-requested distribution. In this case the user-requested sampling distribution still has statistical implications, particularly for the Bayesian interpretation of results, where it plays the role of the prior probability distribution. For example, if the user requests that a parameter have a Gaussian prior $\pi(y)$, and chooses to draw samples with a Metropolis–Hastings algorithm, then the final density of points will be proportional to the posterior probability density $p(y)$

$$p(y) \propto \mathcal{L}(y)\pi(y). \tag{5}$$

This is because it is a property of the Metropolis–Hastings algorithm that the density of sample points is proportional to $\mathcal{L}$ in the unit-interval parameter space – which is then distorted to the physical parameter space density $d(y)$ under the mapping $y = F^{-1}(x)$

$$d(y) = \mathcal{L}(y)\left|\frac{\mathrm{d}F(y)}{\mathrm{d}y}\right| \tag{6}$$

$$= \mathcal{L}(y)f(y). \tag{7}$$

Here $f(y)$ is the probability distribution function (PDF) corresponding to the CDF $F(y)$, and is therefore the user-requested 'prior', and $d(y)$ is proportional to the posterior probability density $p(y)$.

ScannerBit makes a wide range of possible prior transformations available. These priors are separated into three groups: one-dimensional (flat, log, double_log_flat_join, sin, cos, tan, cot), multi-dimensional (gaussian, cauchy), and others (same_as, fixed_value, none, plugin). These priors, and their corresponding options, can be specified in the `Priors` section of the YAML input file that defines a scan, or, in the case of one-dimensional priors, also in the `Parameters` section (see Sect. 4). Users can also define custom priors, which can be added to the set of priors available to ScannerBit (see Appendix C).

3.1.1 Built-in one-dimensional priors

ScannerBit currently includes six one-dimensional priors:

sin: $\mathcal{P}(x) \propto \sin(x)$
cos: $\mathcal{P}(x) \propto \cos(x)$
tan: $\mathcal{P}(x) \propto \tan(x)$
cot: $\mathcal{P}(x) \propto \cot(x)$
flat: Uniform in x, i.e. $\mathcal{P}(x) \propto$ const.
log: Uniform in $\log x$, i.e. $\mathcal{P}(x) \propto 1/x$.
double_log_flat_join: A piecewise prior that patches together sections uniform in $\log(-x)$, uniform in x, and uniform in $\log x$. Useful when the desired prior density is positive at zero, but logarithmic at large absolute values of the parameter. i.e.

$$\mathcal{P}(x) \propto \begin{cases} 1/|x| : & \texttt{lower} < x < \texttt{flat_start} \\ \text{const} : & \texttt{flat_start} \le x \le \texttt{flat_end} \\ 1/x \ : & \texttt{flat_end} < x < \texttt{upper} \end{cases}$$

Each prior has a number of configurable options. These may be entered as key-value entries for the parameter in question, in the input YAML file. For one-dimensional priors, the options can be entered in either the `Priors` or the `Parameters` section of the YAML file (further details on the input file format can be found in Sect. 4). The following options are available for all 1D priors except double_log_flat_join:

`range`: Specifies the range in the form [low, high].
`shift`: Shifts all parameter samples by the specified value. Defaults to `0` if absent.
`scale`: Multiplies all parameter samples by the specified value. If set to `degrees`, will convert degrees to radians. Defaults to `1` if absent.
`output_scaled_values`: If `true`, any scale and/or shift applied to the parameter during a scan is also applied to the printed value of the parameter. Defaults to `true` if absent.

The double_log_flat_join prior also accepts the same `range` option, as well as

`ranges`: An extended version of `range`, taking the form [lower, flat_start, flat_end, upper]. The negative log prior is applied over parameter values ranging from the first to the second entry, the flat prior is applied from the second to the third entry, and the positive log prior is applied between the third and fourth entries. This option takes precedence over `range`.
`flat_start,flat_end`: The boundaries of the interior region over which to apply the flat prior; these options are expected whenever the 4-component `ranges` option is not in use.
`lower,upper`: The outer boundaries of the logarithmic prior sections. These options are only used if neither `ranges` nor `range` is present. They require the presence of `flat_start` and `flat_end`.

3.1.2 Built-in multi-dimensional priors

ScannerBit presently ships with two real multi-dimensional priors, and one example function:

gaussian: Gaussian distribution of the form
$\mathcal{P}(\mathbf{x}) \propto \exp[-(\mathbf{x}-\bar{\mathbf{x}})\cdot C^{-1}\cdot(\mathbf{x}-\bar{\mathbf{x}})/2]$,
with C a covariance matrix.
cauchy: Cauchy distribution of the form
$\mathcal{P}(\mathbf{x}) \propto \left[1+(\mathbf{x}-\bar{\mathbf{x}})\cdot C^{-1}\cdot(\mathbf{x}-\bar{\mathbf{x}})\right]^{-1}$,
with C a covariance matrix.
dummy: Performs a dummy transformation of the unit hypercube parameters back to themselves; included as a simple example of the code needed to define a new multidimensional prior (see Appendix C).

The gaussian and cauchy priors have options:

`cov`: Full covariance matrix. Off-diagonal elements default to zero if this option is omitted.

`sigs`: A vector containing the square root of each of the diagonal components of the covariance matrix. Defaults to `1` if absent.
`mean`: A vector containing the mean (for gaussian) or median (for cauchy) of each parameter. Defaults to `0` if absent.

3.1.3 Additional built-in priors

ScannerBit is also equipped with some useful non-standard priors:

same_as: Specifies that some parameter is the same as another parameter. The net effect is to make both parameters appear as a single parameter to the scanner, but as two distinct parameters to the objective function. This prior accepts an eponymous option `same_as`, which is used to choose which parameter to shadow. It also optionally accepts the `scale` and `shift` keywords described in Sect. 3.1.1, allowing the parameter to be presented to the objective function as a rescaled, shifted version of the parameter it has been set up to shadow.
fixed_value: Fixes this parameter to a specified value, with the actual value set by the option of the same name. If a sequence of values is given, the values are simply iterated over in each subsequent point, repeating from the beginning once exhausted. This prior also accepts the `scale` and `shift` keywords.
none: Specifies that this parameter will be directly set by the scanner. If the scanner does not do so, ScannerBit will throw an error.
plugin: Uses a plugin function as the prior. The plugin to be used is set with an option of the same name (i.e., `plugin`), and must be defined as an objective plugin under the `objectives` tag in the `Scanner` section of the input YAML file. Note that in the current version of ScannerBit, using the same plugin more than once in a given scan is not supported, e.g. as two separate applications of a one-dimensional prior to two different parameters.

3.2 Plugins

ScannerBit plugins are independent code snippets, separate from the main ScannerBit code. **Scanner plugins** provide a standard interface between ScannerBit and sampling algorithms (whether external libraries or native ScannerBit implementations). Objective plugins (otherwise known as **test function plugins**) provide an interface between ScannerBit and external objective or test functions.

Plugin functionality falls into three main categories: loading, unloading, and the main function provided to ScannerBit by the plugin.

loading: When a plugin is loaded, it is provided with some generic information needed for running any plugin, as well as specific information relevant to its plugin type. The generic information includes a list of expected input file options, as well as interfaces to the **printer** and prior transform. Plugin-specific information may include likelihood functor access, hypercube parameter dimension, and parameter key names. Each plugin has a constructor that runs when the plugin is loaded, allowing it to perform startup operations such as variable initialisation.
unloading: When a plugin is no longer needed, any shared libraries it has loaded are unloaded, and the plugin deconstructor runs. This typically performs any plugin-specific shutdown operations, such as closing files or releasing memory.
main function: Every plugin has some core functionality, provided by its `plugin_main` function. For example, a scanner plugin's `plugin_main` should contain code that samples an objective function over a specified parameter space – whereas an objective plugin to be used as a likelihood function would provide functionality necessary for likelihood evaluations. This functionality may have any interface, but it must be consistent with the goal of the plugin. For example, a likelihood plugin should accept a map of parameters and return a likelihood value, whereas a scanner plugin would not accept inputs.

Because of this general format, plugins can be used for a wide range of tasks. Scanner plugins specifically contain code to perform parameter scans of various models, do not require inputs, and simply return an integer indicating the success or failure of the scan. Objective plugins are for more general use, and may provide functions that can be used as likelihoods, observable functions, prior transforms, or in fact any other quantity that might need to be computed for each point in parameter space (e.g. likelihood gradients). Objective plugins are not required to have any specific interface, but are all granted access to the same information and utility functions by ScannerBit. Detailed information about definition, design and operation of ScannerBit plugins can be found in Appendix D.

4 Setup and input file options

ScannerBit scans are specified and initiated using an input file written in YAML. This file must contain at least four sections: `Parameters`, `Scanner`, `Printers` and `KeyValues`. It may also optionally contain a `Priors` section. We do not deal with the `Printers` and `KeyValues` sections in this paper, as they refer to GAMBIT features described in detail in Ref. [29]; minimal working entries for these sections can be found in the example input YAML file given in

Appendix F. Additionally, ScannerBit includes an example YAML file, `ScannerBit.yaml`, in the `yaml_files` folder. The `Parameters` section indicates which models and parameters to scan, as well as (optionally) simple prior definitions for individual parameters. The `Priors` section contains additional – potentially more complicated – prior definitions not included in the `Parameters` section. The `Scanner` section contains all scanner and plugin options and definitions.

4.1 Input file `Parameters` section

The `Parameters` section contains information about the models and their associated parameters, and follows the basic format:

```
Parameters:
  model:
    parameter_name1:
      ...options...
    parameter_name2:
      ...options...
    ...
```

The `Parameters` section can contain several models, where each model contains several parameters. Each declared parameter can have the following options, associated with the prior to be applied to the parameter:

`prior_type`: Specifies a one-dimensional prior to be applied to the parameter. If this option is absent but either the `range`, `same_as` or `fixed_value` option is given, ScannerBit will deduce the prior type from the presence of the other option.
`range`: Specifies the range of parameter values to be sampled. In the absence of an entry for `prior_type`, specifying a `range` causes a flat prior to be adopted.
`shift`: Adds the given value to the parameter.
`scale`: Multiplies the parameter by the given amount.
`same_as`: Indicates that this prior is the same as another parameter. Note that ScannerBit parameters are denoted by a string of the form *model*::*parameter_name*.
`fixed_value`: Fixes the parameter to the given value. The same effect can be achieved in even more compact form, by giving no options for a parameter except a value or sequence of values, in the form *parameter_name*`:` *value*.
`lower,flat_start,flat_end,upper`: for the double_log_flat_join prior (see Sect. 3.1.1).

Each of these options are optional. If none of them is set, the prior must be specified in the `Priors` section. Like the flat prior, the fixed_value and same_as priors do not need to be specifically indicated with `prior_type`, as they are implicitly defined by the declaration of their options. More details can be found in the subsection dealing specifically with one-dimensional priors (Sect. 3.1.1).

4.2 Input file `Priors` section

Any parameter lacking a specified one-dimensional prior in the `Parameters` section must be associated with a sampling range and prior in the `Priors` section. A prior definition in this section takes the form:

```
Priors:
  prior_name:
    parameters: [parameter_list]
    prior_type: type
    options
```

Here, *prior_name* can be any unique identifier, and need not map to any particular name within ScannerBit. The *parameter_list* is a sequence of parameters to apply the prior to. The *type* of the prior must match one of the known ScannerBit priors listed in Sect. 3.1. This should be followed by any additional key-value pairs needed to set the desired *options* of the chosen prior.

4.3 Input file `Scanner` section

The `Scanner` section defines the scanners, objectives and their options. It has the general form:

```
Scanner:
  use_objectives: [objective1, objective2, ...]
  use_scanner: chosen_scanner

  scanners:
    scanner1:
      plugin: plugin1
      options

    scanner2:
      plugin: plugin2
      options

    ...

  objectives:
    objective1:
      purpose: purpose1
      plugin: plugin3
      options

    objective2:
      purpose: purpose2
      plugin: plugin4
      options

    ...
```

All scanners that a user wishes to make available for a given scan must be listed in the `scanners` node, and all objectives in the `objectives` node. Each scanner or objective must be given a local name (*scanner1*, *scanner2*, *objective1*, etc), and a plugin and any relevant options must be associated with that name. Objectives also need to be assigned a `purpose`, which tells ScannerBit and its scanner plugins how the objective

plugin should be used. Exactly one of the scanners under the `scanner` node can be chosen as the sampling algorithm for the scan, by setting `use_scanner` to the name of the block that defines the preferred scanner. Arbitrarily many objectives can be activated with the `use_objectives` directive.

4.4 ScannerBit standalone executable

Like other GAMBIT modules, ScannerBit can be compiled into a standalone executable, and used independently of GAMBIT. This can be useful for sampling external objective functions that do not come from GAMBIT. The build command is simply

```
make ScannerBit_standalone
```

which creates the executable ScannerBit_standalone and places it in the main GAMBIT directory.

The interface of the ScannerBit _standalone is similar to that of GAMBIT itself. Launching `ScannerBit_standalone -f` *yaml_file* runs a scan defined in the file *yaml_file*. To replace rather than resume from any existing files when beginning a scan, use the `-r` option.

ScannerBit_standalone also provides a diagnostic list of recognised scanners and objective plugins à la GAMBIT, using the commands `ScannerBit_standalone scanners` and `ScannerBit_standalone objectives` (or simply `ScannerBit_ standalone plugins` to see both together). These commands list the name, version, and status of all the plugins that ScannerBit is aware of.

The standalone can also provide diagnostic information on a specific plugin, using the command `ScannerBit_standalone` *plugin_name*. Individual plugin diagnostics contain three sections. The *General Plugin Information* section displays the name, type, version, and status of the plugin. The status `ok` indicates that a plugin is properly linked. The status `reqd lib(s) not found` indicates that a library requested by the `reqd_libraries` macro cannot be found. A status of `invalid lib path(s) in locations file` indicates that a library specified in `config/scanner_locations.yaml` or `config/objective_locations.yaml` (or their default equivalents; see Sect. D.1) cannot be found at the specified location. Similarly, `reqd header file(s) not found` occurs when a header listed under `reqd_headers` cannot be located, and `invalid include dir(s) in locations file` indicates that an include folder that was specified in the `scanner_locations.yaml` or `objective_locations.yaml` files cannot be located. Finally, a status of `excluded` indicates that the plugin was `-Ditch`ed from the configuration of the code when CMake was invoked. The *Header & Link Info* section contains include and link paths of headers and libraries requested by the plugin, information about which of them were found, and a list of all input file options that the plugin requires to be defined in order to run. Finally, the *Description* section contains a short description of the plugin. This typically includes recognised input file options and a description of the algorithm or function that the plugin provides.

5 Simple scanners

ScannerBit includes four simple scanners, all found in `ScannerBit/src/scanners/simple/`: a random sampler, a grid sampler, a list-based raster sampler, and a simple toy Metropolis MCMC toy_mcmc. These are all parallelised with MPI, using a simple prescription that simply distributes objective calculations evenly among the available processes. Below we give the available options for each simple scanner, and default values in square brackets (where defaults exist).

5.1 The random sampler

The random sampler draws a user-defined number of random points from the specified prior. The only available option is

`point_number[10]`: The number of random samples desired.

5.2 The grid and square_grid scanners

These scanners calculate likelihoods at points on a uniform, user-defined grid in the unit hypercube. The grid scanner allows the grid resolution be specified separately for each parameter, whereas square_grid is simply a shortcut for the special case where the grid has the same number of points in every dimension. The grid resolution is set with the option

`grid_pts[2]`: For the grid scanner, a vector of integers that specifies the number of grid points in each dimension of the parameter space. For the square_grid scanner, a single integer.

5.3 The raster scanner

This scanner computes an objective over a user-defined list of parameter points. The available options are:

`like`: The purpose to use as the objective.
`parameters`: The parameters specified by the user.

The `parameters` option should contain a list of parameters, with a number or sequence that specifies the user-defined values, e.g.

```
raster_example:
```

```
plugin: raster
like: LogLike
parameters:
  "model::param_1": [0, 1]
  "model::param_2": 0.5
  "model::param_3": [2, 3, 4]
```

To obtain sensible results, the none prior should be employed for any parameters where values are given via the `parameters` option. Any parameters not specified are chosen randomly, and transformed by the chosen priors. Parameters can be specified with a single number to apply to all points in the list, or as a vector of values. Different parameters can be assigned lists of different lengths, which simply repeat once they are exhausted. In the example above, ScannerBit will run the points $(0, 0.5, 2) \rightarrow (1, 0.5, 3) \rightarrow (0, 0.5, 4)$, and then terminate.

5.4 The toy_mcmc scanner

This the simplest possible implementation of the Metropolis algorithm [37], with the proposal distribution set to the prior. Given a randomly drawn initial point x_i, a candidate point x_i' is randomly selected from the unit hypercube. The candidate is then accepted with probability

$$\alpha = \min[1, \mathcal{L}(x_i')/\mathcal{L}(x_i)]. \tag{8}$$

If a point is accepted, it becomes the comparison point in the next iteration. If it is rejected, the previous point is retained. The scanner keeps track of the number of times a given point is retained, and the resulting point multiplicities can then be used as weights in subsequent analysis, in particular for computing Bayesian posterior probability densities. There is no convergence criterion implemented in the toy_mcmc; the scanner simply runs for a fixed number of points given by the user:

`point_number[1000]`: The number of distinct (accepted) points to be computed in the chain.

6 The postprocessor

This plugin reads a series of samples computed in some previous scan, and computes additional likelihoods or observables for them. Log-likelihoods for the original samples may be added to or subtracted from a newly-computed contribution, allowing existing likelihood constraints to be replaced or new ones added to previously-completed scans. Like the simple scanners, the postprocessor uses MPI to divide its objective calculations evenly between available processes.

The postprocessor operates as a scanner plugin. From the perspective of ScannerBit and GAMBIT, it *is* a scanning algorithm. However, it does not generate sample points for itself, but instead obtains them directly from previous scan output. When running from GAMBIT, this means that the **likelihood container** then operates using the parameter values from the previous scan as input, and the output likelihood and observables are added to the existing data from the previous scan. A new set of output files is created, just as they are when running a 'true' scan. All data from the original output that does not conflict with new output is copied to the new output files, leaving the original files unchanged.

In most respects, the postprocessor operates as a standard GAMBIT scanner: it can be run via the standard GAMBIT interface, it can be run in parallel via MPI, it can be stopped and resumed, and all **printer** output from the likelihood container is treated the same as it would be during a 'normal' scan. The options and particulars of the postprocessor are given in Appendix B.1.

7 Markov Chain Monte Carlo

In Bayesian parameter estimation and model comparison, calculating evidence values or one-dimensional posterior PDFs for individual parameters or observables requires the ability to integrate the full multi-dimensional posterior density. An efficient sampling method for the posterior PDF is therefore mandatory. Of the methods proposed for this task, Markov Chain Monte Carlo (MCMC) algorithms are amongst the most tried and tested [38,39].

In general, MCMC methods allow one to study any target distribution of a vector of parameters $\boldsymbol{\theta}$, by generating a sequence of n parameter combinations (a 'chain') $\{\boldsymbol{\theta}_i\}_{i=1,\ldots,n} = \{\boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \ldots, \boldsymbol{\theta}_n\}$. The chain constitutes a Markov process, because each $\boldsymbol{\theta}_{i+1}$ is drawn from a proposal distribution that is fully determined by the previous point $\boldsymbol{\theta}_i$. MCMC algorithms are designed to ensure that the time spent by the Markov chain in a region of the parameter space is proportional to the target posterior PDF value in this region. Hence, from such a chain, one can obtain a series of independent samples from the posterior PDF. Up to a common normalisation constant (the evidence), both the target posterior PDF and any marginalised versions of it can be estimated by simply counting the number of samples within the relevant region of parameter space.

7.1 The GreAT software

The Grenoble Analysis Toolkit (GreAT) [27] is a modular, user-friendly, object-oriented C++ MCMC framework for sampling user-defined parameter spaces. It uses the Metropolis-Hastings algorithm [37–40] to generate Markov chains. This prescription ensures that the stationary distribution of the chain asymptotically tends to the target distri-

bution (typically the posterior PDF), by generating a candidate state $\boldsymbol{\theta}_{\rm trial}$ picked at random from a proposal distribution $q(\boldsymbol{\theta}_{\rm trial}|\boldsymbol{\theta}_i)$ and accepting the candidate with probability a,

$$a(\boldsymbol{\theta}_{\rm trial}|\boldsymbol{\theta}_i) = \min\left(1, \frac{p(\boldsymbol{\theta}_{\rm trial})}{p(\boldsymbol{\theta}_i)}\frac{q(\boldsymbol{\theta}_i|\boldsymbol{\theta}_{\rm trial})}{q(\boldsymbol{\theta}_{\rm trial}|\boldsymbol{\theta}_i)}\right). \tag{9}$$

Here, the target distribution $p(\boldsymbol{\theta})$ can be reduced to the likelihood function $\mathcal{L}$ assuming a flat prior for $\boldsymbol{\theta}$. If the trial is accepted, it becomes the new state, whereas if it is rejected, the current state is retained. This criterion ensures that once at its equilibrium, the chain samples the target distribution $p(\boldsymbol{\theta})$.

To optimise the efficiency of an MCMC, the proposal distribution should be as close as possible to the true distribution. The MCMC implemented in GreAT uses a multivariate Gaussian distribution, accounting for possible correlations between the parameters of the model. GreAT runs multiple MCMC chains, either sequentially or in parallel depending on the user's MPI configuration. At the termination of each chain, based on the samples contained in all chains completed so far, i.e. minus a 'burn-in' period at the beginning of each chain and after the removal of correlated samples by thinning the chains, GreAT updates the covariance matrix to be used to define the proposal distribution in subsequent chains. The updated covariance matrix is saved externally, in order to allow chains running in parallel to always use the latest version.

To obtain a reliable estimate of the target distribution, GreAT bases its analysis of a chain on a selected subset of its points. Burn-in points are discarded, to avoid the random starting point of the chain biasing the sampling. By construction, each step of the chain is correlated with the previous steps: GreAT obtains sets of independent samples by thinning the chain over its autocorrelation length l. The single-parameter autocorrelation on length scale k, in a chain of total length N and for parameter θ, is

$$r(k) = \frac{\sum_{i=1}^{N-k}(\theta_i - \bar{\theta})(\theta_{i+k} - \bar{\theta})}{\sum_{i=1}^{N}(\theta_i - \bar{\theta})^2}. \tag{10}$$

GreAT defines the correlation length l_j for the jth parameter to be the smallest inter-sample interval such that $r(l_j) \leq 0.5$; samples separated by scales larger than this are considered independent. The overall correlation length l for the chain is defined as the maximum correlation length across all m parameters, i.e. $l \equiv \max_{j=1..m} l_j$.

The fraction of independent samples measuring the efficiency of the MCMC is defined to be the fraction of samples remaining after discarding the burn-in steps and thinning the chain. The final results of the MCMC analysis are the target distribution and all marginalised distributions, obtained by counting the number of samples within the relevant region of parameter space.

7.2 GreAT–ScannerBit interface

As implemented in GreAT, the Metropolis–Hastings algorithm has no default convergence criterion. The user is required to specify a chain length, i.e. a number of steps, for each Markov chain. These options are given in Appendix B.2

GreAT also extracts the relevant trials for further analysis. It first calculates the burn-in length b corresponding to the first sample $\boldsymbol{\theta}_b$ for which $p(\boldsymbol{\theta}_b) > p_{1/2}$, where $p_{1/2}$ is the median of the target distribution obtained from the entire chain (i.e. the median posterior density, at least in standard applications). To obtain uncorrelated samples within each chain, it then computes the autocorrelation function for each parameter (Eq. 10). If the chain does not have any samples for which $p(\boldsymbol{\theta}_b) > p_{1/2}$, then a burn-in length cannot be defined. This can happen if e.g. every sample has the same likelihood, as can occur if every sample in the chain is deemed invalid by ScannerBit, and assigned the default minimum log-likelihood (set by the YAML entry `KeyValues::likelihood::model_invalid_for_lnlike_below`).

GreAT performs these operations after computing each chain, before using the results to update the covariance matrix. If the burn-in length of the last chain is undefined, a warning message is printed, and a new chain is started using the old covariance matrix. Chains for which the burn-in length is undefined are not retained for any further analysis, and are considered invalid. At the end of the run, the complete statistics for all valid chains (burn-in length, correlation length, number of independent samples) are printed out in GreAT's native format. The independent samples and their multiplicities are stored in whatever output format the user has instructed GAMBIT to use for printing results.

8 Ensemble MCMC

Standard MCMC algorithms are traditionally somewhat problematic in large or highly multi-modal parameter spaces, as their efficient operation requires a well-tuned proposal density. Some modern MCMC samplers (such as GreAT) address this by adaptively varying the proposal distribution based on samples from previous runs. Other successful strategies use multiple concurrent MCMC chains as the basis of the proposal distribution. These are commonly referred to as ensemble samplers.

In an ensemble MCMC, each chain is individually advanced by constructing a proposal PDF from the set of all current points across the full set of concurrent chains.

Procedurally, this equates to exploring an augmented parameter space consisting of n copies of the original space, $\{\boldsymbol{\theta}_{(0)}, \boldsymbol{\theta}_{(1)}, \ldots, \boldsymbol{\theta}_{(n)}\}$ corresponding to a composite posterior distribution $\mathcal{P}(\boldsymbol{\theta}_{(0)}, \boldsymbol{\theta}_{(1)}, \ldots, \boldsymbol{\theta}_{(n)}) = \prod_{i=0}^{n} \mathcal{P}(\boldsymbol{\theta}_{(i)})$ where $\mathcal{P}(\boldsymbol{\theta})$ is the actual target distribution of interest. These algorithms are able to easily adapt their proposal densities to the target distribution, and exhibit performance that is generally invariant under affine transforms (e.g. $\boldsymbol{\theta} \rightarrow \phi\boldsymbol{\theta}$). Unfortunately, the performance of these algorithms is highly sensitive to the number of concurrent chains, with the number of chains required typically scaling linearly with the parameter dimension; this makes the overall number of likelihood evaluations needed for convergence proportional to the square of the parameter dimension.

8.1 T-Walk

In the serial version of the T-Walk algorithm [41], chains are advanced one at a time, with the proposal density based on the current parameter points of all chains not chosen for advancement, and the chain to be advanced chosen randomly at each iteration. In the parallel version, each MPI process randomly selects a chain for advancement at each iteration, and the proposal distribution used for advancing all chains is based only on the state of the remaining chains not chosen for advancement by any process in that iteration. In what follows, we refer to chains that are being advanced in a given iteration as the advancing chains, and the others (those contributing to the proposal distribution) as the proposal chains.

T-Walk uses one of four movement strategies when advancing a chain, choosing randomly between them at each iteration. Two of these strategies, the walk and traverse moves, shift the current chain position ($\boldsymbol{\theta}_i$) by some multiple of the distance between it and the current point in a randomly-selected proposal chain. The remaining two moves, hop and blow, cause advancing chains to perform different random Gaussian jumps, with covariance matrices calculated from the full set of current points in the proposal chains.

Walk: advances the current chain $\boldsymbol{\theta}_\mathbf{i}$ by jumping either towards or away from a randomly selected proposal chain $\boldsymbol{\theta}_j, i \neq j$. This move produces a candidate point $\boldsymbol{\theta}'_i$,

$$\boldsymbol{\theta}'_i = \boldsymbol{\theta}_i + (1-\alpha)(\boldsymbol{\theta}_j - \boldsymbol{\theta}_i), \tag{11}$$

where α is a parameter drawn from a distribution $\mathcal{G}(\alpha)$. For distributions satisfying

$$\mathcal{G}\left(\frac{1}{\alpha}\right) = \alpha\mathcal{G}(\alpha), \tag{12}$$

detailed balance is satisfied if the candidate point is accepted with probability

$$p = \min\left[1, \alpha^{n-1}\frac{\mathcal{P}(\boldsymbol{\theta}'_i)}{\mathcal{P}(\boldsymbol{\theta}_i)}\right]. \tag{13}$$

Here n is the dimension in which the T-Walk moves are being performed (the so-called 'projection dimension', described later in this subsection). ScannerBit's implementation of T-Walk uses the distribution

$$\mathcal{G}(\alpha) = \frac{\sqrt{a_w}}{2(a_w-1)} \times \begin{cases} \frac{1}{\sqrt{\alpha}}, & \text{for } \frac{1}{a_w} \leq \alpha \leq a_w \\ 0 & \text{otherwise,} \end{cases} \tag{14}$$

where a_w is a user-configurable input parameter of the algorithm.

Traverse: similar to walk, but the chain is advanced by jumping *over* the point in the proposal chain. The candidate point is

$$\boldsymbol{\theta}'_i = \boldsymbol{\theta}_i + (1+\beta)(\boldsymbol{\theta}_j - \boldsymbol{\theta}_i), \tag{15}$$

where β can take any positive value. Detailed balance is satisfied if β follows a distribution $\mathcal{H}(\beta)$ that satisfies

$$\mathcal{H}\left(\frac{1}{\beta}\right) = \mathcal{H}(\beta), \tag{16}$$

and the Metropolis-Hastings acceptance probability is modified as

$$p = \min\left[1, \beta^{n-2}\frac{\mathcal{P}(\boldsymbol{\theta}'_i)}{\mathcal{P}(\boldsymbol{\theta}_i)}\right], \tag{17}$$

where n is again the projection dimension. ScannerBit's implementation of T-Walk uses

$$\mathcal{H}(\beta) = \frac{a_t^2-1}{2a_t} \times \begin{cases} \beta^{a_t} & \text{for } 0 < \beta \leq 1 \\ \beta^{-a_t} & \text{for } \beta > 1, \end{cases} \tag{18}$$

where a_t is another parameter of the algorithm, configurable by the user.

Hop and blow: In general, the walk and traverse moves available to the advancing chains only form a basis for some smaller-dimensional subspace of the full parameter space. With only these moves available, if the current chain positions are co-planar or are sufficiently clustered, mixing between chains can be low, and infinite loops of identical repeated and reversed jumps can occur. For this reason, traditional MCMC jumps are mixed into the proposal distribution. These moves use the total set of current points in the proposal chains to infer a covariance matrix C. The hop and blow moves use C to construct a

Gaussian proposal function and perform an MCMC jump based on the resulting conditional PDF; hop centers the proposal on the current point of the chain being advanced, whereas blow centers it on the current point of one of the proposal chains.

ScannerBit's implementations of hop and blow advance a chain some distance r in a chosen direction $\hat{\mathbf{r}}$ from the center of the proposal distribution. Following [6], r is drawn from the distribution

$$\mathcal{P}(r) = \frac{2}{3}\mathcal{P}_2(r/d) + \frac{1}{3}e^{-r}, \tag{19}$$

where $\mathcal{P}_n(x)$ is the distribution of radii arising from an n-dimensional normal distribution centred at the origin, and d is the user-configurable Gaussian jump parameter. The distance r is related to the hypercube parameters $\boldsymbol{\theta}$ via a Cholesky decomposition $C = \mathbf{L}\mathbf{L}^T$,

$$\boldsymbol{\theta}_i' = \boldsymbol{\theta}_k + r\mathbf{L}\cdot\hat{\mathbf{r}}. \tag{20}$$

The starting point $\boldsymbol{\theta}_k$ of the jump for the hop move is the current point of the chain to be advanced, whereas the starting point of the blow move is the current point of any other advancing or proposal chains.

In order to promote exploration of the parameter space in scenarios where the best-fit regions are highly degenerate in the parameters, T-Walk chooses the direction of propagation $\hat{\mathbf{r}}$ by first choosing a random orthonormal basis for the parameter space. It then chooses $\hat{\mathbf{r}}$ in successive hop and blow moves by cycling through the basis vectors in random order. Once it has used all basis vectors once, it generates a new random orthonormal basis.

T-Walk calculates C directly from the current points of the proposal chains,

$$C = \sum_j \left(\boldsymbol{\theta}_{(j)} - \bar{\boldsymbol{\theta}}\right)\left(\boldsymbol{\theta}_{(j)} - \bar{\boldsymbol{\theta}}\right)^T, \tag{21}$$

where j indexes the proposal chains, and $\bar{\boldsymbol{\theta}}$ gives the mean current point across them. If this matrix is not positive-definite, then T-Walk approximates it as

$$C_{l,l} = \left(\max_{j,k}\left[\theta_{l(j)} - \theta_{l(k)}\right]\right)^2/12 \tag{22}$$

where j and k run over all proposal chains.

ScannerBit's implementation performs each walk and traverse step within a randomly chosen subspace of lower dimensionality, known as the projection subspace. This encourages chain movement by avoiding a narrow distribution, which is endemic to higher-dimensional proposal distributions. The relative probabilities of walk and traverse moves are set equal, as are those of hop and blow. The ratio of walk+traverse to hop+blow moves, and the dimension of the projection subspace, are user-configurable.

The version of the T-Walk algorithm described above, and implemented in ScannerBit, differs slightly from the original algorithm [41] in two ways. The first is the use of the full concurrent covariance matrix for the Gaussian jumps in the hop and blow moves, making them similar to the "walk" move of Ref. [42]. Second, the algorithm is formulated to work with any number of chains greater than one, rather than just a pair (making the walk and traverse moves described here similar to the "stretch" move in Ref. [42]).

The version of T-Walk in ScannerBit uses the Gelman-Rubin convergence diagnostic $\sqrt{R}$ [43] to determine convergence. This statistic compares the inter-chain dispersion to the total dispersion of each parameter.

See Appendix B.3 for the available options and outputs of T-Walk.

9 Nested sampling

Nesting sampling is a method designed for efficient calculation of the Bayesian evidence. As a byproduct, it also produces samples from the posterior. The algorithm samples the posterior in nested shells of probability, by continually updating a set of "live" points, replacing the lowest-likelihood live point in each iteration with a better point. As the algorithm progresses, the set of live points naturally splits into clusters that shrink around the peaks of the posterior, making the algorithm well-suited to efficiently sampling multimodal distributions. MultiNest [17] is a Fortran library that implements the nested sampling algorithm, with the addition of a clustering algorithm to estimate bounding ellipsoids for the live points. These bounding ellipsoids are used to approximate the iso-likelihood contours of the function being explored, allowing the algorithm to efficiently propose new live points when scanning parameter spaces of low to moderate dimension. For large dimensionalities the MultiNest algorithm is computationally expensive, as the bounding ellipsoids typically encompass large swathes of uninteresting parameter space – but for small and moderate-size parameter spaces it usually offers quite competitive efficiency. The ScannerBit plugin runs the MultiNest sampler developed by Feroz et al. [17]. Its options and outputs are listed in Appendix B.4.

10 Differential evolution

Differential evolution [44–47] (DE) is an efficient algorithm for global optimisation, with similarities to both genetic algorithms and the Nelder–Mead simplex method [46]. It has been

found to be quite robust, and is often the algorithm of choice for multimodal, high-dimensional problems.

DE works by evolving a population of points in parameter space, with successive generations chosen by a form of vector addition between members of the current population. The vector addition step gives the algorithm the character of a random walk with a step size provided by the population. This makes it highly adaptive, and helps to limit the number and tuning of control parameters required. In its simplest form, DE requires only three controlling parameters; this can be reduced even further in variants that allow self-adaptation of parameters. It is straightforward and efficient to parallelise, as each member of the population can be simultaneously and independently evaluated against a replacement candidate.

DE's population-based mutation also leads to *contour matching* [48], where members of a population will tend to be at similar likelihood values, with the worst individuals improving the fastest, allowing the algorithm to trace out contours of the objective function rather effectively. This not only allows good mapping of likelihood contours, but further aids with adaptive stepping from one generation to the next, and promotes transfer of population members between local minima, improving the overall convergence towards the global minimum.

10.1 Algorithmic details

All variants of DE consist of three main steps: mutation, crossover, and selection. These are controlled by three parameters: the population size NP, the mutation scale factor F, and the crossover rate Cr. The simplest form of DE, known as 'rand/1/bin', was first described in 1995 [44], and continues to be widely used. The first two parts of the name refer to the strategy for mutation, and the third refers to the crossover; these are described in detail below.

The algorithm begins by initialising the population to a random selection of points within the allowed parameter space (Fig. 1). We will denote the population of points (also referred to as *target vectors*) as $\{\mathbf{X}_i^g\}$, with i indexing the members of the population, and g indexing the generation. Each subsequent generation of the population is chosen by performing mutation, crossover and selection on the previous generation.

10.1.1 Mutation

The first step in DE is mutation, which will produce the *donor vectors* $\{\mathbf{V}_i\}$ from the current population of target vectors $\{\mathbf{X}_i^0\}$. This step is illustrated in Fig. 2. In the rand/1 mutation scheme, a random vector is combined with a single difference vector scaled by the mutation scale factor F. To produce each donor vector $\mathbf{V}_i$, three random vectors $\mathbf{X}_{r1}$, $\mathbf{X}_{r2}$ and $\mathbf{X}_{r3}$ are chosen from the current population, such that none of the

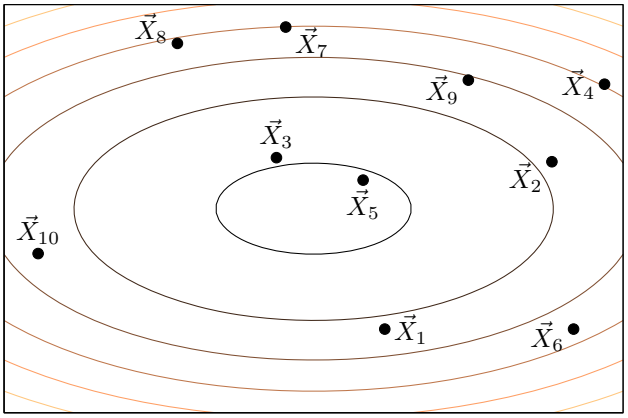


Fig. 1 A simple example of differential evolution in two dimensions. This figure shows the likelihood function represented by contours (with more central contours corresponding to higher likelihood values), and an initial random population of NP = 10 vectors $\{\mathbf{X}_i^0\}$. Subsequent figures illustrate the remaining steps of the algorithm

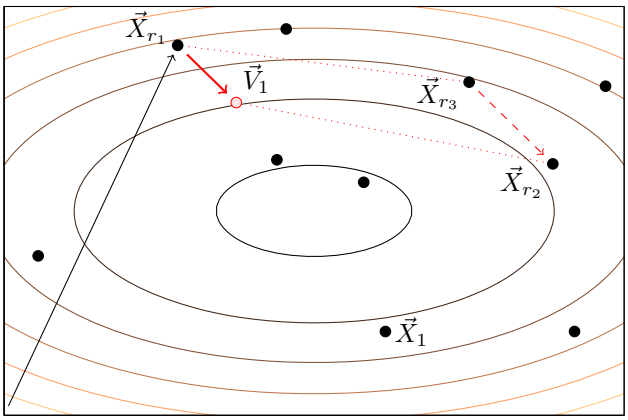


Fig. 2 The process of creating the first donor vector during mutation in the simple 'rand/1' variant of this step. The difference vector between two randomly chosen points is shown as a dashed red line, and the scaled difference vector (thick red line) is shown added to another randomly chosen point to create the donor vector $\mathbf{V}_1$. Note that the current target vector $\mathbf{X}_1$ is not used during rand/1 mutation. The scale factor in this example is F = 0.7. Ellipses are isolikelihood contours, with more central contours corresponding to higher likelihood values

$\mathbf{X}_k$ are the same, and none matches the current target vector $\mathbf{X}_i$. The vectors are then combined using vector addition to produce the donor vector:

$$\mathbf{V}_i = \mathbf{X}_{r1} + F(\mathbf{X}_{r2} - \mathbf{X}_{r3}). \tag{23}$$

This name rand/1 refers specifically to the fact that the donor is formed by choosing a *random* base vector from the population, and vector-adding it to *one* scaled difference vector between population members. The combination of a single target vector (referred to as the *base vector*) with a donor vector constructed from scaled differences between other population members is a general feature of DE. Further variants are detailed in Sect. 10.1.4.

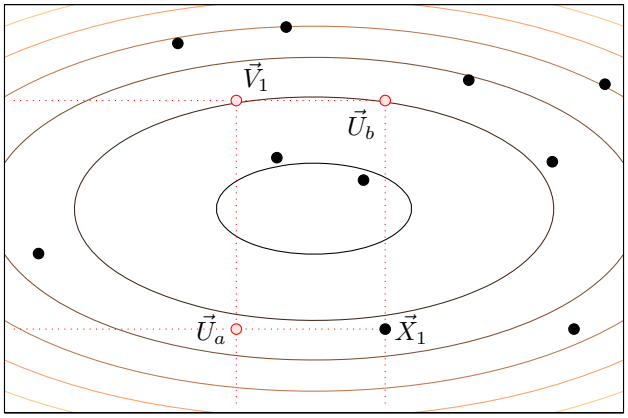


Fig. 3 Binomial crossover between the donor vector $\mathbf{V}_1$ and the target vector $\mathbf{X}_1$ in rand/1/bin differential evolution. This produces three possible trial vectors, shown in lightly-filled red circles. Because at least one component of the donor vector always goes into the trial vector, but no components are guaranteed to come from the target vector, $\mathbf{V}_1$ is a possible trial vector (in the case where both components have been taken from the donor vector), as are $\mathbf{U}_a$ and $\mathbf{U}_b$ (where only one component has been chosen from the donor vector). The target vector $\mathbf{X}_1$ itself is not a possible trial vector. Ellipses are isolikelihood contours, with more central contours corresponding to higher likelihood values

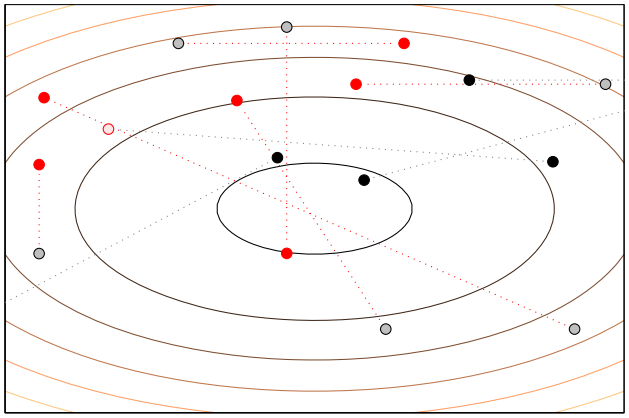


Fig. 4 The last step in a generation of differential evolution. This shows the process of selection after trial vectors have been chosen for the entire population. Each target vector is compared with its associated trial vector, and the better one is retained for the next generation. Here red indicates trial vectors and black indicates target vectors. Filled circles have been kept for the next generation, whereas open circles have been rejected. Note that several points have trial vectors outside the allowed boundaries; these are rejected automatically. Ellipses are isolikelihood contours, with more central contours corresponding to higher likelihood values

The usage of this vector addition strategy allows DE to explore a function dynamically, based on the size and shape of the evolving population (which reflects the size and shape of the contours of the objective function). The value of F is the main determinant of how broad this search is. In general, F is required to be less than 1 for convergence to be achievable – but too low a value can lead to insufficient exploration, and premature convergence [48].

10.1.2 Crossover

The second step in DE is crossover, also called recombination. This is illustrated in Fig. 3. Crossover combines the donor vectors produced by mutation with the original population of target vectors to produce the *trial vectors* $\mathbf{U}_i$. The trial vectors will potentially form the next generation of vectors. The degree to which the trial vectors are composed of components of the donor vectors rather than components of target vectors is influenced by the parameter Cr, which takes a value between 0 and 1. In binomial crossover (the 'bin' of rand/1/bin DE), the trial vector is chosen according to the following procedure:

1. For the kth component of the trial vector $\mathbf{U}_i$, denoted $\mathbf{U}_{i,k}$, a random number r_k is chosen such that $0 < r < 1$.
2. If $r_k \leq Cr$, the component is taken from the donor vector: $\mathbf{U}_{i,k} = \mathbf{V}_{i,k}$.
3. If $r_k > Cr$, it is taken from the target vector instead: $\mathbf{U}_{i,k} = \mathbf{X}_{i,k}$.
4. After all components of $\mathbf{U}_i$ have been chosen in this fashion, one component is reassigned in order to ensure that trial vectors are always different from their parent target vectors. A dimension l is chosen randomly for each member of the population. The corresponding component of the donor vector is then assigned to the target vector: $\mathbf{U}_{i,l} = \mathbf{V}_{i,l}$, irrespective of its previous value.

As Cr increases, the probability that components are chosen from the donor vector increases: for many-dimensional problems, the percentage of components taken from the donor vector is approximately Cr (see Ref. [49] for a full analysis). High values of Cr therefore lead to increased exploration, as the trial vectors will differ from the target vectors along many dimensions. Low values of Cr are primarily effective for the special case where the likelihood function is a separable function of the parameters, because this allows the algorithm to explore along individual dimensions [e.g. 48]. In the more general case, where the objective function is non-separable, Cr should be kept high to allow better exploration. A small amount of crossover with the target vectors remains useful, however, as it improves the diversity of the population of trial vectors [48].

10.1.3 Selection

The final step in DE is selection, which generates the next population of vectors. This step is shown in Fig. 4. The value of the objective function (typically the likelihood) for each target vector $\mathbf{X}_i^g$ (the previous population) is compared with

the trial vector $\mathbf{U}_i$ constructed from it using mutation and crossover. The point with the better likelihood is retained as a member of the next generation, and becomes one of the new target vectors $\mathbf{X}_i^{g+1}$. If both have the same likelihood, the trial vector $\mathbf{U}_i$ is preferred, in order to allow the population to move across flat surfaces.

Selection makes DE what is known as a *greedy* algorithm: it takes any improvement offered, and never accepts steps that would lead to a poorer fit. This allows faster convergence, but unlike non-greedy sampling methods (e.g. MCMCs), where poorer fits are sometimes accepted, discovery of the global minimum is not guaranteed even for infinite running time.

It is possible for trial vectors to be located outside of the allowed parameter space boundary. This is most common during the first few generations of the algorithm, when the population is spread out, allowing very large difference vectors to be produced. However, if a local or global minimum is located near the edges of parameter space, out of bounds vectors can occur throughout the minimisation process. The simplest way to enforce parameter boundaries is to reject any trial points that lie outside them; for alternatives see Sect. A.2.

10.1.4 Advanced mutation and crossover strategies

Although rand/1/bin DE is simple and popular, many other variants have been proposed. The simplest variations involve either a different choice of base vector, or a different method to calculate the difference vector. The name of the DE strategy is typically written in the form *base/difference number/crossover*, where

base: how the base vector, $\mathbf{X}_{r1}$ in Eq. 23 and Fig. 2, is chosen for mutation.
difference number: the number of difference vectors $F\,(\mathbf{X}_{r2} - \mathbf{X}_{r3})$ in Eq. 23 and Fig. 2 that are used in mutation.
crossover: the form of crossover used.

Some options for the base vector beyond a random choice from the population include the current target vector ('current'), the best vector in the population ('best'), or a base vector made up of a combination of these (e.g. 'rand-to-best').

A 'general' mutation strategy encompassing several possible mutation strategies can be written as follows [50]:

$$\mathbf{V}_i = \lambda \mathbf{X}_{\text{best}} + (1-\lambda)\mathbf{X}_1 + \sum_{q=0}^{Q} F_q(\mathbf{X}_{2_q} - \mathbf{X}_{3_q}), \tag{24}$$

where $\mathbf{X}_1$ is the current vector or is chosen randomly as before and $\mathbf{X}_{2,3}$ are chosen randomly from the population. No vectors may be used twice. This form allows rand base vectors ($\mathbf{X}_1 = \mathbf{X}_{\text{rand}}$ and $\lambda = 0$), current base vectors ($\mathbf{X}_1 = \mathbf{X}_i$ and $\lambda = 0$), best base vectors ($\lambda = 1$), rand-to-best base vectors ($\mathbf{X}_1 = \mathbf{X}_{\text{rand}}$ and $0 < \lambda < 1$), and current-to-best base vectors ($\mathbf{X}_1 = \mathbf{X}_i$ and $0 < \lambda < 1$). It also allows for the use of Q difference vectors along with a corresponding set $\{F_q\}$ of scale factors. Note that there are other forms of mutation that are not described by this equation.

Using the best individual in the population as the base vector (e.g. best/1/bin) speeds up convergence, as it reduces stagnation in the population – but it makes DE less likely to find the global minimum compared to simply choosing the base randomly. This tends to be a good choice for near-unimodal functions, but poor for highly multimodal functions [48,51]. Using the current vector as the base can slow convergence because it reduces the diversity of the resulting population [48], but can be more efficient than randomly choosing the base because it reduces so-called 'selection drift bias' [47]. Combining multiple difference vectors can help combat the loss of diversity induced by using either the best or current vector as the base, but may hamper contour-matching [48].

In contrast to the proliferation of mutation strategies, binomial crossover has only one main competitor, exponential crossover ('exp'). The lack of additional recipes is mostly a result of the lesser impact of crossover on performance than mutation [52]. Exponential crossover was used in the original DE algorithm [44], but is generally less popular than binomial crossover.

In exponential crossover, a length L to be crossed over is chosen by drawing random numbers between 0 and 1 until one of them exceeds Cr. L is then set to the total number of draws required, with the provision that it must be less than the dimensionality of the parameter space D. A random dimension d is then chosen from $[1, D]$, and the next L entries in the donor vector (wrapping around to the first if necessary) are chosen to contribute to the trial vector. The remaining $D - L$ components are taken from the target vector.

Exponential crossover is generally considered to perform less well than binomial crossover. This has been suggested [51] to be due to the requirement in exponential crossover that dimensions taken from the target vector must be adjacent, whereas in binomial crossover all combinations are possible. Both forms of crossover suffer from the fact that the process is not rotationally invariant, as it preferentially acts along dimensions, and therefore cannot perform identically on separable and inseparable functions, decreasing efficiency when working with parameterisations that induce correlations between parameters [48,52]. This is a common feature of evolutionary algorithms, including e.g. genetic algorithms.

10.1.5 Self-adaptive differential evolution

As with all optimisation strategies, the ideal choice of parameters for DE depends on the type of problem to be solved, and is frequently unclear *a priori*. The ability for the algorithm to

adapt its parameters in real time is therefore advantageous. One example of self-adaptive differential evolution is known as jDE [53], which compares favourably with classic DE and other modifications of DE across problem types and in high-dimension parameter spaces [46,54].

The jDE algorithm is based on classic rand/1/bin DE but adapts the values of F and Cr as the run progresses. Each vector in the population is associated with personal values of F and Cr, which are then used to generate the next generation of vectors. Before mutation occurs for the ith member of the population, F_i has a chance to change. The same is true of Cr_i immediately before crossover. During selection, the values of F and Cr belonging to successful vectors are retained in the next generation of the population. Variants on the jDE algorithm can extend the self-adaptive behaviour to other mutation or crossover strategies. We introduce one such variant, λjDE, which dynamically modifies λ in a similar way over the course of the run. We describe the jDE and λjDE algorithms, as well as our implementations and variations of them, in greater detail in Sect. 10.2.2.

10.2 The Diver package

In this section, we introduce Diver, an open-source differential evolution sampler intended for use in optimisation problems in physics and astronomy. Diver can be downloaded either as a source tarball or a git repository from http://diver.hepforge.org. It is released under an academic use license.

10.2.1 Design and invocation

Diver is a fully-featured, standalone parallel implementation of differential evolution. Its default mode is to perform self-adaptive λjDE optimisation, with jDE, rand/1/bin and all mutation and crossover strategies in between available through an extensive set of runtime options. It also includes additional options for outputting derived parameters, stopping and restarting scans, computing approximations to various Bayesian quantities, and dealing with discrete parameters.

Diver is written in Fortran, and includes wrappers for calling it from C/C++. It is compatible with gcc 4.4 and later, and version 11 and later of the intel compiler suite. Parallelism in Diver makes use of MPI, and works by simply dividing each generation up evenly across all MPI processes. It is invoked by calling the Fortran function `diver()` or its C equivalent `cdiver()` from some user-supplier driver program. When calling these functions, the driver program must pass the address of another, user-supplied, likelihood/objective function, which Diver then minimises. The package includes example driver programs and objective functions in Fortran, C and C++; these can be respectively found in the `example_f`, `example_c`, and `example_cpp` subdirectories of the main Diver installation directory.

Synopses of the different source files in Diver, the various run options it offers, and the format of its outputs can be found in Appendices A.1, A.2 and A.3, respectively.

10.2.2 Adaptive differential evolution: jDE and λjDE

We include two options to use self-adaptive evolution, based on the jDE algorithm initially proposed by Brest et al. [53]. In regular jDE (accessed by setting `jDE = true`), rand/1/bin evolution is used, but each vector has unique values for F and Cr, which evolve along with the population.

The evolution of F is controlled by a value τ_1, which we take to be 0.1 throughout. The permissible range for F extends from $F_l = 0.1$ to $F_u = 0.9$, as values of F too close to zero imply no evolution, whereas values too close to 1 prevent convergence. We choose the initial value of F for each vector randomly from a uniform distribution between F_l and F_u. Before mutating the vectors, we draw a random number and compare it to τ_1. If it less than τ_1, we update F to a new random value between F_l and F_u, and the new value is used for mutation. Then, during selection, if the trial vector is accepted, the new value for F is kept as well. If the trial vector is rejected, the previous value for F is kept instead.

Similarly, the evolution of Cr is controlled by a value τ_2, also taken to be 0.1. Unlike F, Cr is allowed to vary between 0 and 1 inclusive, as crossover does not exhibit any pathological behaviour in either limit. For each member of the population, we initialise Cr to a random value between 0 and 1. For each generation, before crossover we then choose a trial value for Cr. As for F, we draw a uniform random deviate and compare it to τ_2; if it is larger than τ_2, the trial value for Cr remains unchanged; if it is smaller, we choose a random new value for Cr and use it during crossover. During selection, if the trial vector is kept, the new crossover parameter is kept as well; if not, the value of Cr reverts to the previous value.

The justification for this process is that different values of F and Cr are useful for different classes of problems, but the preferred values are usually not known. It is presumed that successful choices of F or Cr are more likely to lead to successful trial vectors, and so by tying the evolution of F and Cr to the evolution of the vectors, desirable values of F and Cr will be preferentially propagated.

In addition to the standard jDE, we offer the possibility to use self-adaptive rand-to-best/1/bin evolution. This works just as in jDE, but with the addition of an adaptive λ mutation parameter, which evolves via a scheme that mirrors the way Cr is evolved. The addition of this parameter harnesses the benefits of jDE, while allowing for more aggressive optimisation, since information about the position of the best member

of the current generation is used. This option is accessed by setting `lambdajDE = true`.

10.2.3 Discrete parameters and parameter-space partitioning

Diver offers the ability to label one or more parameters as discrete rather than continuous, using the `discrete` keyword. This may be desirable because some parameter(s) are indeed discrete at some fundamental level, or simply as a means of labelling a set of individual fits that are interrelated in some way.

The main complication when working with discrete parameters is that mutation must be a floating-point operation in DE, in order to ensure that the donor vectors are valid, to allow for enough variety in potential donor vectors, and to ensure proper convergence. When treating a parameter as discrete in Diver, we deal with this by storing the values of the discrete parameter internally as floating-point values, so that mutation works as normal, but evaluation of the likelihood is done by rounding the parameter to the closest integer. The output `.raw` file stores the underlying floating-point representation of the parameters (to allow runs to be properly resumed), whereas the desired integer values are output in a `.sam` file (we discuss output formats in more detail in Appendix A.3).

The `partitionDiscrete` option can also be used to partition the DE population evenly into the allowed values of the discrete parameters. With this option, no vector is allowed to change its discrete value. This mode allows simultaneous fitting of multiple objective functions, with the discrete dimension simply treated as a label for assigning subpopulations to the different problems. One useful application of this option is to perform multi-objective optimisation where the value of each fitness function depends (preferably only weakly) on the best-fit parameters of the other subpopulations.

10.2.4 Population diversity and duplicate individuals

In order for DE to converge appropriately, it is necessary to retain sufficient population diversity. Duplicate vectors in the population lead to artificial drops in diversity. Duplicate vectors can arise naturally in rand/ or best/ mutation if two separate vectors in the population are updated using the same combination of random vectors. Once there are multiple identical vectors in a population, the diversity of the population will decrease, making premature convergence more likely.

Even more problematically, duplicate vectors have a tendency to infect the rest of the population: whenever a pair of duplicates is chosen to create the difference vector during mutation, the resulting donor vector will match the third vector chosen, possibly creating another duplicate. In best/ mutation, such a process can rapidly lead to an entire population matching the 'best' vector.

Diver includes a facility for weeding out duplicate vectors as soon as they arise to prevent these problems. When `removeDuplicates = true`, the population is examined after selection. If a set of duplicates is discovered, one is modified, according to the following rules:

1. If one vector was inherited from the previous generation, and the other is new, the new vector is reverted to its previous value.
2. If both vectors are new, the one that improved the most is kept and the other is reverted to its previous value.
3. The appearance of duplicate vectors in the initial population, or inheritance of multiple copies of the same vector from a previous generation, are strong indications of coding errors. In these cases, a warning is printed and one vector is re-initialised to a random point in the parameter space.

Duplicate removal is disabled by default for current/ mutation (`current = true`), jDE (`jDE = true`), and λjDE (`lambdajDE = true`), as the presence of duplicates in the results of these algorithms would be surprising. It is enabled by default for all other settings, i.e. rand/, best/, or rand-to-best/ mutation, as these forms of mutation are susceptible to duplicate creation. If Diver is compiled with MPI support, duplicate removal is enabled by default regardless of any other settings, and is recommended as a useful diagnostic for insuring against MPI library issues.

10.2.5 Approximate posterior and evidence estimates

Diver can compute the Bayesian posterior and evidence from its samples when using a negative log-likelihood function as the objective, by using the likelihood samples to perform Monte Carlo integration of the (prior-weighted) likelihood. These calculations can be activated by setting `doBayesian` = true and specifying a `prior` function.

Because DE does not share the property of Bayesian algorithms that the sampling distribution is proportional to the posterior, this requires a bootstrap estimate of the actual sampling distribution produced in a DE run. This invariably leads to fairly rough estimates of Bayesian quantities, especially when the likelihood function is multimodal and/or highly non-Gaussian, but the results can be useful for some quick estimates before deploying more expensive algorithms optimised for Bayesian inference.

Diver obtains a bootstrap estimate of its sampling density by performing a binary space partitioning on the parameter space being scanned, using the actual samples obtained in a scan. Each sample is sorted into a cell in the partitioned parameter space, with cells partitioned further as

soon as their populations exceed `maxNodePop`. The partitioning is done alternately in each direction of the parameter space, so that each cell remains rectangular in the parameters.

The resulting posterior weight for a sample θ can then be estimated as

$$P(\theta) \approx \frac{N_c}{N_s} V(\theta) \Pi(\theta) \mathcal{L}(\theta), \tag{25}$$

where N_c is the number of cells, N_s the total number of samples, $V(\theta)$ is the parameter volume occupied by the cell containing the sample θ, $\Pi(\theta)$ is the prior function (provided explicitly by `prior` – note that this is *not* the prior transform, but the prior itself), and $\mathcal{L}(\theta)$ is the likelihood, i.e. $\exp(-x)$, where $x \equiv -\ln \mathcal{L}$ is the objective function being sampled. The corresponding Monte Carlo estimate of the Bayesian evidence is then

$$\mathcal{Z} \approx \sum_{i=1}^{N_s} P(\theta_i). \tag{26}$$

Taking the estimate to be Gaussianly distributed, the 1σ uncertainty on the evidence can be approximated from its variance,

$$\Delta\mathcal{Z} \approx \sqrt{(\langle P^2 \rangle - \mathcal{Z}^2)/N_s}, \tag{27}$$

where

$$\langle P^2 \rangle \equiv \frac{1}{N_s} \sum_{i=1}^{N_s} P^2(\theta_i) \tag{28}$$

is the mean square posterior.

If `doBayesian = true`, Diver will continue to sample until the logarithmic uncertainty on $\mathcal{Z}$ reaches or passes below `Ztolerance`, i.e.

$$\ln\left(\frac{\mathcal{Z}}{\mathcal{Z} - \Delta\mathcal{Z}}\right) \leq \texttt{Ztolerance}. \tag{29}$$

Once this convergence criterion has been satisfied, Diver then further polishes its posterior and evidence estimates by taking the final binary spanning tree so generated during the scan, and re-calculating Eq. 25 for each individual of every population. This improves the final posterior and evidence estimates because the resulting weights for all individuals get computed on the basis of the complete tree, rather than the tree as it was at the time each individual was initially created.

10.2.6 ScannerBit interface

Because Diver is specifically designed to minimise positive-definite fitness functions, the Diver plugin for ScannerBit uses the *negative* of the composite log-likelihood function provided by GAMBIT as its fitness function. If desired, ScannerBit will also apply an offset to the log-likelihood passed to Diver, and have the printer remove that offset again before printing. This can be useful in cases where the likelihood normalisation leads to positive total log-likelihoods; taken without an offset, these likelihoods would prevent the fitness passed to Diver from remaining positive definite. The offset can be specified with the `lnlike_offset` option in the `likelihood` node of the `KeyValues` section of a run's main YAML file. If this option is absent, the offset will default to 10^{-4} times the value of `model_invalid_for_lnlike_below` (also in `KeyValues::likelihood`). The full range of Diver options available from the YAML file is given in Appendix B.5.

The Diver interface in ScannerBit does not yet make use of the ability of Diver to scan discrete parameters, as doing so is not yet supported by ScannerBit itself; this feature is slated for inclusion in a future revision of GAMBIT.

11 Scanner performance comparisons

By offering the capacity to vary the scanning algorithm and its operating parameters – whilst keeping all other aspects of a scan identical – ScannerBit provides a unique testbed for comparing sampling algorithms. In this section we present an exploration of the performance of the four major scanners available in GAMBIT 1.0.0, when applied to a physically realistic likelihood function. The modularity of the scanner interface allows consistent comparison between both the algorithms themselves, and between different choices of algorithm parameters.

This investigation is intended to reveal the strengths and weaknesses of different sampling algorithms with respect to typical user requirements. These requirements can be quite varied, and may include the choice of statistical approach (frequentist or Bayesian), the time taken for a scan to converge, the reliability of the results, or some combination of the three. However, for any thorough investigation, the user should typically take advantage of the unique flexibility offered by ScannerBit to employ a range of algorithms, statistical methods, and scanner parameters in order to obtain the most complete and robust sampling possible.

For this demonstration, we work with the scalar singlet dark matter model. This model has two parameters beyond the Standard Model (SM): the Higgs portal coupling λ_{hS}, and

Table 1 Parameters, ranges and central values of the test scans of this section, for each scan dimensionality. The ranges for most SM parameters correspond to $\pm 3\sigma$ variations around the 2014 PDG central values [63]. For the Higgs, the range is $\pm 4\sigma$ about the 2014 central value (which encompasses the 2015 4σ range [64]). For the up and down quark masses, we take the central values from the 2014 review, and scan over a range of $\pm 20\%$ around the central values. This is intended to capture the $\pm 3\sigma$ range implied by the likelihoods in PrecisionBit [31], which deal with correlated mass-ratio measurements. The nuclear couplings also incorporate a range of $\pm 3\sigma$ around the best estimates. The dark matter density has an asymmetric range about the central value, as the likelihood that we apply to this parameter is log-normal rather than Gaussian. We refer the reader to Refs. [35,57] for further details and references on the central values and uncertainties associated with the local density and nuclear parameters

Parameter		Values
Scalar pole mass	m_S	$[45, 10^4]$ GeV
Higgs portal coupling	λ_{hS}	$[10^{-4}, 10]$
Varied in 7 and 15-dimensional scans		
Electromagnetic coupling	$1/\alpha^{\overline{MS}}(m_Z)$	127.940(42)
Strong coupling	$\alpha_s^{\overline{MS}}(m_Z)$	0.1185(18)
Top pole mass	m_t	173.34(2.28) GeV
Higgs pole mass	m_h	125.7(1.6) GeV
Local dark matter density	ρ_0	$0.4^{+0.4}_{-0.2}$ GeV cm^{-3}
Varied in 15-dimensional scans		
Nuclear matrix el. (strange)	σ_s	43(24) MeV
Nuclear matrix el. (up + down)	σ_l	58(27) MeV
Fermi coupling $\times 10^5$	$G_{F,5}$	1.1663787(18)
Down quark mass	$m_d^{\overline{MS}}(2\,\text{GeV})$	4.80(96) MeV
Up quark mass	$m_u^{\overline{MS}}(2\,\text{GeV})$	2.30(46) MeV
Strange quark mass	$m_s^{\overline{MS}}(2\,\text{GeV})$	95(15) MeV
Charm quark mass	$m_c^{\overline{MS}}(m_c)$	1.275(75) GeV
Bottom quark mass	$m_b^{\overline{MS}}(m_b)$	4.18(9) GeV

the singlet Lagrangian mass parameter μ_S. We present the results in the effective parameter space of λ_{hS} and m_S, where the physical singlet mass m_S is given by

$$m_S = \sqrt{\mu_S^2 + \frac{1}{2}\lambda_{hS} v_0^2} \tag{30}$$

where $v_0 = 246$ GeV is the vacuum expectation value of the Higgs field. The likelihood and posterior are both multimodal and highly degenerate across several orders of magnitude in the values of these parameters.

To investigate how performance scales with dimensionality, we introduce additional parameters that enter into the combined likelihood function. These parameters are well constrained by unimodal likelihood functions, but still create a significant challenge for any sampling algorithm due to the increase in the dimensionality of the parameter space. In particular, we carry out detailed tests in two, seven and fifteen dimensions, and one scan with each sampler for dimensionalities between two and fifteen. We list the free parameters for each scan in Table 1. For all test scans, we apply a logarithmic prior to the singlet parameters λ_{hS} and m_S, and flat priors to the additional parameters.

In the following, we only show full results from the fifteen-dimensional scans. Increasing the dimensionality of the problem across this particular parameter space does not substantially shift the location nor shape of the final likelihood with respect to λ_{hS} and m_S. As a result, the best-fit point and regions of maximum likelihood remain similar. For comparison, in Appendix E, we give additional detailed results in two dimensions. The inclusion of additional parameters does significantly increase the runtime for the scanning algorithms, and degrades their ability to locate the maximum likelihood point. Note that choosing a more complicated model, with more complicated parameters in the 'higher' dimensions, would only increase the required computing time, making such an extensive comparison study infeasible. We refer the interested reader to the companion papers on supersymmetric models [33,34] for applications of Diver and MultiNest to higher-dimensional multimodal parameter spaces.

The dominant physical constraints on the model that we consider here come from experiments searching for dark matter via direct and indirect detection, the observed limit on the thermal relic abundance of dark matter, and constraints on the rate of invisible Higgs decays at the Large Hadron Collider. We also apply the constraint $\lambda_{hS} < 10$, as larger

values would violate perturbative unitarity and are therefore not physically interesting. More details on the model can be found in accompanying and earlier papers [29,35,55–62]. Here our test function consists of the same likelihood components as in Ref. [35]. Although this is a simple, well-studied extension of the SM, the parameter space is still sufficiently non-trivial that it constitutes an illustrative test of scanner performance.

In Sects. 11.1–11.4 we discuss the most appropriate choices of settings for MultiNest, Diver, T-Walk and GreAT, respectively. In order to make comparisons, we require fair metrics with which to compare the outcomes of scans. We first look at the best value of the log-likelihood found in each scan, which is crucial for the correct normalisation of the profile likelihood (Figs. 5, 6, 10 and 13). The results of this test favour algorithms primarily intended as optimisers, whilst disadvantaging those mainly designed to map the likelihood function or posterior. We therefore also compare the visual quality of the profile likelihood maps (Figs. 7, 9, 11 and 14), and the corresponding posterior maps (Figs. 8, 12 and 15). This is a more qualitative approach, better suited for algorithms intended to explore the parameter space.

We also make some additional comparisons between the four sampling algorithms. In the first two of these tests, we are interested in the relative performance as a function of parameter space dimensionality (Sect. 11.5) and the total CPU time required to complete a scan (Sect. 11.6). Here, we focus mostly on the value of the best-fit log-likelihood and the time taken to achieve it. These sections are most relevant for evaluating profile likelihood performance; in Sect. 11.7, we instead focus on the specific merits of different algorithms for mapping the Bayesian posterior. We discuss the overall implications of these results in Sect. 11.8.

We performed all tests using a high-performance computing cluster, taking advantage of the ability to run GAMBIT in parallel across multiple processors. In the interests of making sensible use of computing resources and time, we ran the two-dimensional scans on a single 24-core compute node, using 24 MPI processes. For the seven- and fifteen-dimensional scans, we used 10 nodes, for a total of 240 MPI processes. For the scans where we compare performance with respect to dimensionality, a consistent computing environment is required; here we used 5 nodes for all scans, corresponding to 120 MPI processes.[3] The two-dimensional profile likelihood and marginalised posterior maps that we show in the following subsections were produced with pippi [36], using 150 bins in each dimension.

[3] Although GAMBIT is also able to use OpenMP threads for further (likelihood-level) parallelisation within individual MPI processes [29], here we limit ourselves to distributed-memory parallelisation with MPI, seeing as this is the form of parallelisation employed by the scanning algorithms.

11.1 MultiNest

MultiNest's ability to accurately evaluate the evidence and map the posterior is directly affected by the number of live points used in a scan, with more live points increasing the chance of finding all relevant modes of the posterior. On the other hand, more live points means more likelihood evaluations, and requires greater computing resources. The overall duration of the scan is also influenced by the stopping criterion, which is given by the tolerance on the final evidence (the estimate of the largest evidence contribution that can be made with the remaining portion of the posterior volume). The sampling parameters that we vary are therefore the number of live points (N_{live}, `nlive`) and the tolerance (`tol`).

We perform runs with 2000, 5000, 10,000 and 20,000 live points, and tolerances of 10^{-4}, 10^{-3}, 10^{-2} and 10^{-1}. The values of the best-fit log-likelihoods achieved for scans using these parameters are shown in Figs. 5 and 6. In Fig. 7, we present a selection of the profile likelihoods from MultiNest scans in the full 15-dimensional parameter space; in Fig. 8 we give corresponding marginalised posterior maps.

We see consistent best fits from all scans when `tol` $\leq 10^{-3}$. A sufficiently small tolerance appears to provide a good best-fit value over a large range of `nlive` values. On the other hand, even with larger values of `nlive`, setting `tol` too large will still negatively impact the quality of the best-fit point; even with 20,000 live points we still see a poor best-fit likelihood if the tolerance is greater than 10^{-3}. The number of live points has a more significant impact on the sampling of the parameter space, as can be seen in Figs. 7 and 8. In these plots, a significant difference in the quality of both profile likelihood and posterior sampling is evident even between runs done with 2000 and 5000 live points.

On the basis of these results, we recommend an upper bound on the tolerance of 10^{-3} if MultiNest is to be relied upon for obtaining the appropriate normalisation for profile likelihoods. The number of live points required will depend on the desired quality of the resultant profile likelihood or posterior contours, and the dimensionality of the parameter space. In Fig. 7, it is clear that in fifteen dimensions a value of at least 20,000 for `nlive` is required to give fine-grained sampling of the profile likelihood. Because in most cases one is interested in a global fit over many parameters, we recommend a value of 20,000 live points as the lower limit. We note however that this may be reduced somewhat if dealing with a lower-dimensional parameter space, or if one is only interested in mapping the posterior at a lower resolution (less bins) than we have employed here.

11.2 Diver

Diver is a differential evolution optimisation package that is also highly effective at sampling parameter spaces. The size

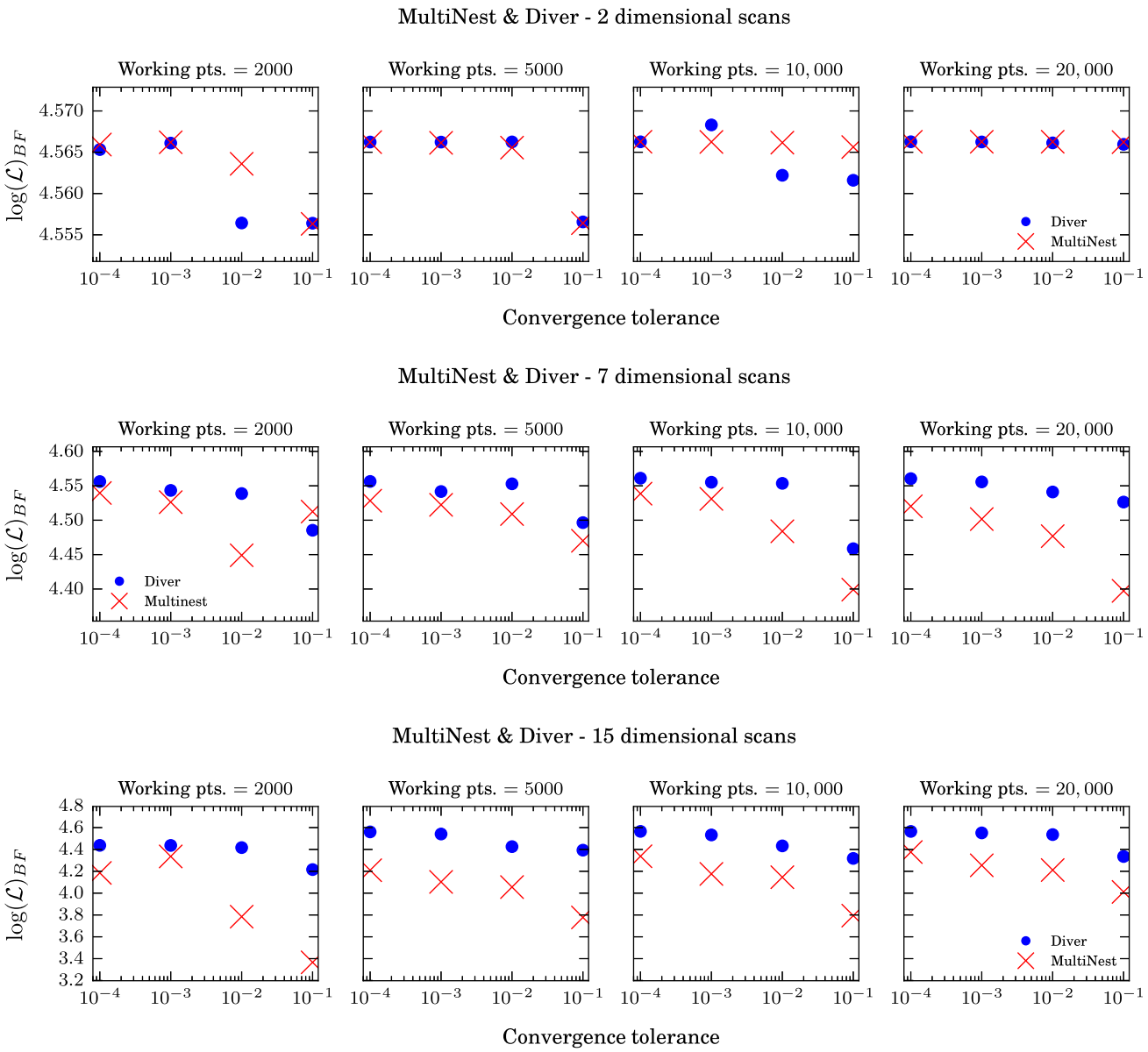


Fig. 5 Best-fit log-likelihoods in scans of the scalar singlet space using the Diver and MultiNest scanners, for a range of convergence tolerances and a fixed number of working points. Tolerances correspond to the parameter `tol` for MultiNest and the parameter `convthresh` for Diver. Working points correspond to the parameter N_{live} for MultiNest and the parameter `NP` for Diver. Note that the likelihood is dimensionful, leading to $\mathcal{L}_{\text{BF}} > 1$ [29]

of the evolving population is determined by the `NP` parameter, and the threshold for convergence is controlled by the `convthresh` parameter.

We examine population sizes of `NP` = 2000, 5000, 10,000 and 20,000, and `convthresh` values of 10^{-4}, 10^{-3}, 10^{-2} and 10^{-1}. Although these parameters have different definitions to `nlive` and `tol` in MultiNest, we take advantage of the similarity in the appropriate ranges for these and plot the scan results on the same axes in Figs. 5 and 6. We see that a `convthresh` value of less than 10^{-3} gives consistent results for the best-fit log-likelihood at all values of `NP`.

In two dimensions, both MultiNest and Diver are able to find roughly the same or equivalently good best-fit points. The differences in the algorithms become evident in seven and fifteen dimensions however, where Diver consistently outperform MultiNest for equivalent parameter values. This is somewhat expected, given that Diver is designed as an optimisation routine, whereas MultiNest is intended to compute

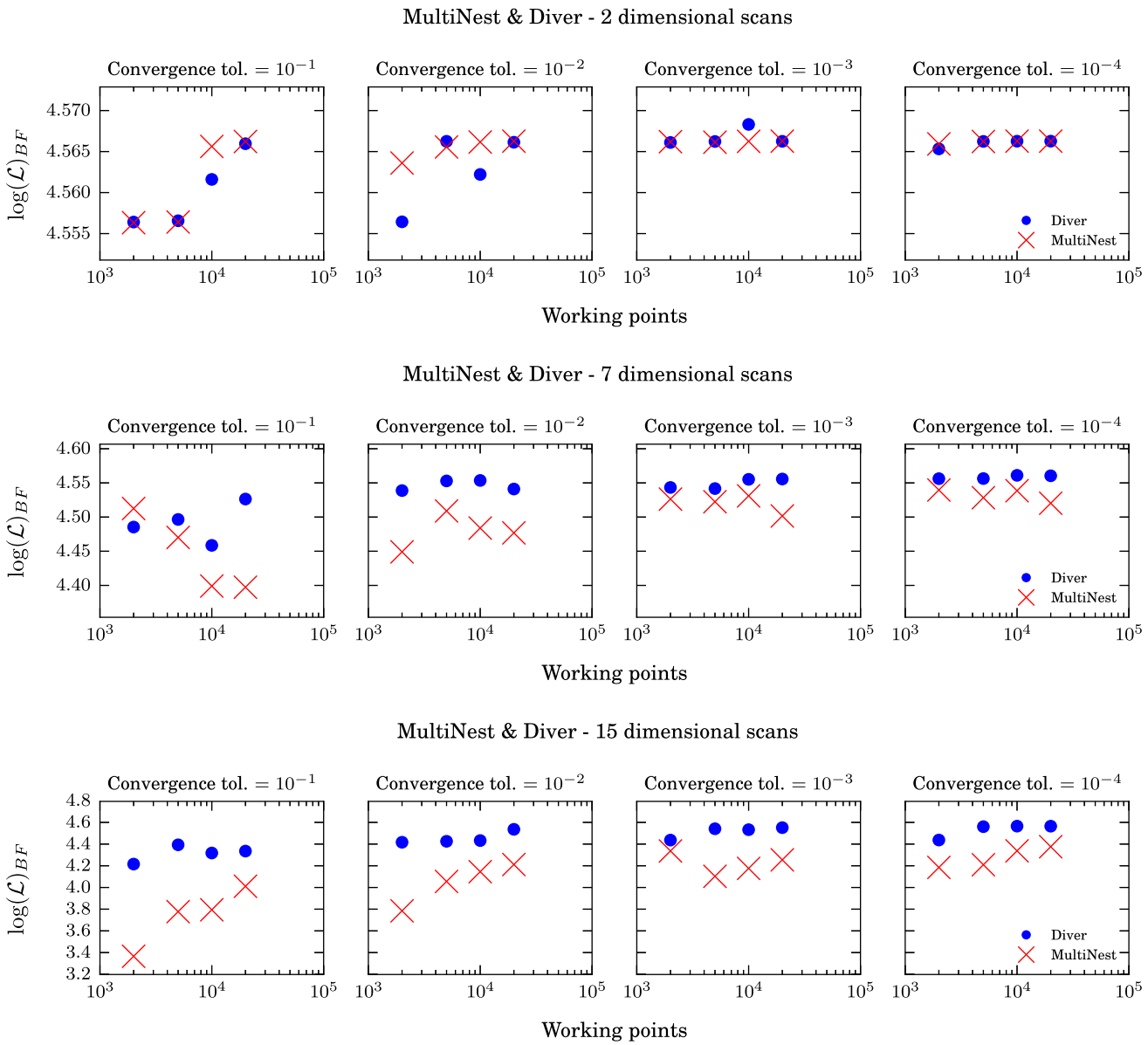


Fig. 6 Best-fit log-likelihoods in scans of the scalar singlet space using the Diver and MultiNest scanners, for different numbers of working points and fixed convergence tolerances. Working points correspond to the parameter N_{live} for MultiNest and the parameter `NP` for Diver. Tolerances correspond to the parameter `tol` for MultiNest and the parameter `convthresh` for Diver. Note that the likelihood is dimensionful, leading to $\mathcal{L}_{\text{BF}} > 1$ [29]

the Bayesian evidence and sample the posterior distribution. In two dimensions, the sampling is dense enough that MultiNest has been able to locate the best-fit point, but in higher dimensions the task is more suited to an optimisation-specific routine. Because the maximum likelihood is located in the low-mass region in both two and fifteen dimensions, it is indeed a result of poor sampling that MultiNest has not located the same best fit that Diver has achieved (see Appendix E for equivalent plots for two dimensional scans). We return to this discussion in Sect. 11.8.

In Fig. 9, we investigate the ability of Diver to accurately map the contours of the profile likelihood. We see that both the `convthresh` and `NP` settings are relevant in reproducing the desired contours. A `convthresh` of 10^{-3} appears appropriate in fifteen dimensions, along with an `NP` value of at least 20,000. However, these requirements become less stringent in a lower-dimensional parameter spaces (data not shown),

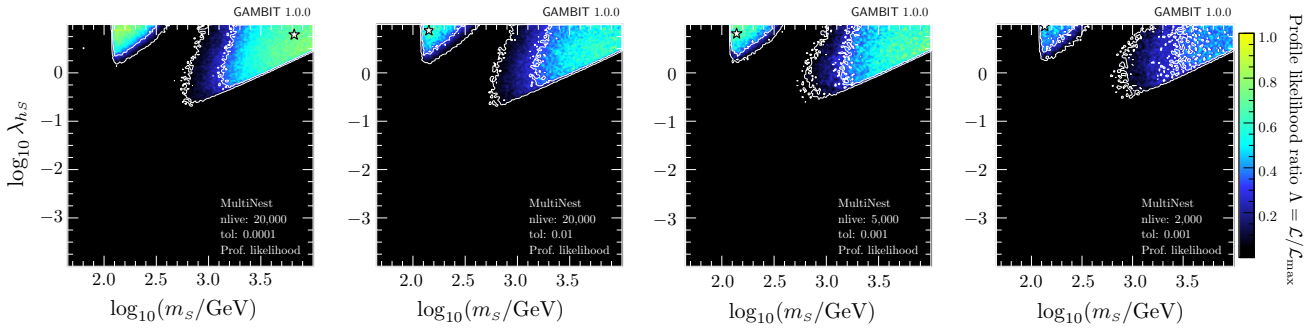


Fig. 7 Profile likelihood ratio maps from a 15-dimensional scan of the scalar singlet parameter space, using the MultiNest scanner with a selection of difference tolerances (`tol`) and numbers of live points (`nlive`). The maximum likelihood point is shown by a white star

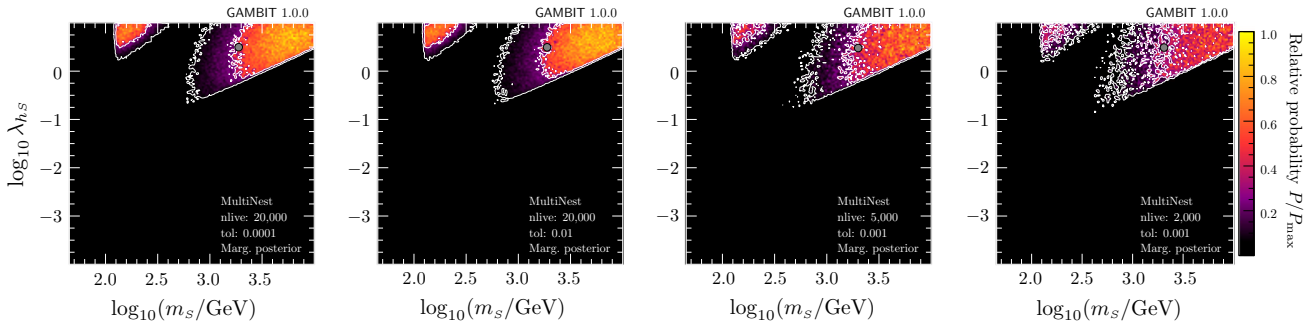


Fig. 8 Marginalised posterior probability density maps from a 15-dimensional scan of the scalar singlet parameter space, using the MultiNest scanner with a selection of difference tolerances (`tol`) and numbers of live points (`nlive`). Note that the colourbar strictly only applies to the rightmost panel, and that colours map to the same enclosed posterior mass on each plot, rather than to the same iso-posterior density level (i.e. the transition from red to purple is designed to occur at the edge of the 1σ credible region, and so on). The posterior mean is shown with a grey bullet point

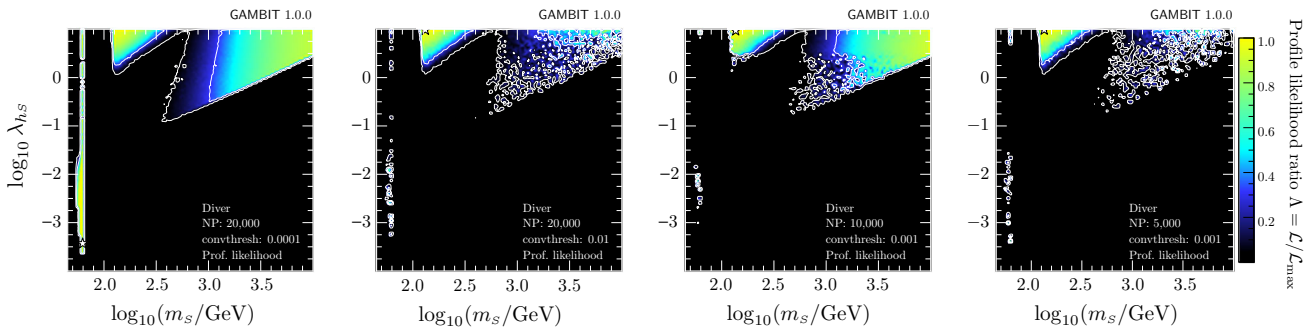


Fig. 9 Profile likelihood ratio maps from a 15-dimensional scan of the scalar singlet parameter space, using the Diver scanner with a selection of difference convergence thresholds (`convthresh`) and population sizes (`NP`). The maximum likelihood point is shown by a white star

where they can be reduced by at least an order of magnitude whilst still achieving a suitable mapping of the profile likelihood.

From these tests, we recommend similar settings as for MultiNest for similar parameters: for a detailed picture of the profile likelihood a value of 20,000 is recommended for `NP` (although this can be reduced for lower dimensional parameter spaces), and to consistently find the best-fit point an upper bound of 10^{-3} is recommended for the `convthresh` convergence tolerance.

11.3 T-Walk

T-Walk is an ensemble MCMC algorithm. The primary parameters of interest are the number of chains used during the scan and the stopping criterion. The latter is controlled by the parameter `sqrtR`, which is the square root of the Gelman-Rubin R statistic, where 1 is perfect. For comparison with other scanners, we define the equivalent tolerance of T-Walk scans as $tol \equiv$ `sqrtR` -1. The `chain_number` is bounded below by 1 + `projection_dimension` + the number of MPI

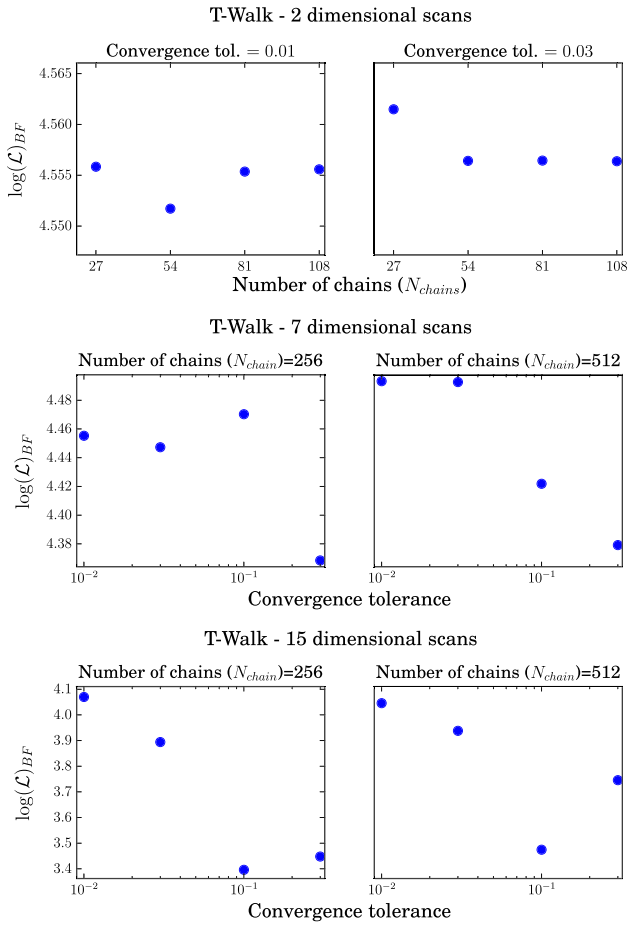


Fig. 10 Top row: Best-fit log-likelihoods for two-dimensional scans using the T-Walk algorithm, as a function of the number of chains used, for two different convergence tolerances (`tol`). Middle and bottom panels: Best-fit log-likelihoods as a function of convergence tolerance (`tol`), for T-Walk scans in seven and fifteen dimensions with a fixed number of chains. Note that the likelihood is dimensionful, leading to $\mathcal{L}_{\rm BF} > 1$ [29]

processes in use (see Sect. B.3). For two dimensions, we have a lower limit of 27 $(24 + 2 + 1)$, and therefore perform tests with 27, 54, 81 and 108 chains. For higher-dimensional scans, the increase in the number of MPI processes requires larger chain numbers, so we choose 256 and 512. We consider tol values of 0.3, 0.1, 0.03 and 0.01.

The best-fit log-likelihoods from scans using various T-Walk settings are given in Fig. 10. In two dimensions, we hold the tolerance fixed and investigate the effect of varying the chain number. We see no notable trend with chain number, for either of the tolerance values. For the seven and fifteen-dimensional scans, we therefore instead focus on varying the tolerance for a fixed number of chains. This reveals the expected trend: smaller tolerances result in improvements to the best-fit log-likelihoods. A significant improvement seems to occur when $tol \lesssim 0.1$. We also notice no significant difference between the scans with 256 and 512 chains, consistent with what we saw in the two-dimensional scans.

In Fig. 11, we show a selection of profile likelihood maps of the 15-dimensional scalar singlet parameter space. We immediately see that smaller tolerances are preferable for a detailed sampling, and doubling the number of chains has no notable impact on the quality of the sampling. In Fig. 12, we show a selection of the marginalised posterior maps of the 15-dimensional scalar singlet parameter space achieved by T-Walk. Here we see that whilst the main posterior modes appear to be better explored with smaller values of `tol`, leading to smoother, better-converged posterior contours, the presence of the minority mode at low mass would seem to be more evident in scans using a *higher* tolerance. This may appear counter-intuitive; why should poorer sampling apparently do better at uncovering small regions such as this? In reality, this region has been sampled more carefully in the scans with lower `tol` values, despite appearing less prominently in the posterior maps. That the sampling in these regions is better at lower tolerances can be seen from Fig. 11, where lower tolerances pick up better-fit points in this region. Nevertheless, the additional samples retrieved in runs with lower tolerances provide a steadily more accurate indication of relative posterior weights of each of these modes, gradually leading to the low-mass solution to become reweighted and disfavoured in the better-sampled posterior maps of Fig. 12.

Recommending parameters for the T-Walk algorithm is difficult, due to the sensitivity of the convergence to the $tol =$ `sqrtR` -1 parameter. However, values less than ~ 0.1 appear to be safe for the scans we have conducted here. Increasing the number of chains above the minimum value does not appear to result in any improvement in the quality of the best-fit, nor in the overall sampling. As starting values for a study using the T-Walk scanner, we therefore recommend setting $tol < 0.1$ and leaving `chain_number` at the default (minimum) value.

11.4 GreAT

The Grenoble Analysis Toolkit (GreAT [27]) is a traditional Metropolis-Hastings MCMC able to sample parameters in parallel using multiple independent chains. The number of chains is controlled by the `nTrialLists` parameter, and the number of points to run each chain for is controlled by `nTrials`. No other convergence criteria are available.

For all dimensionalities, we consider `nTrials` values of 100, 200, 500, 1000, 2000, 5000 and 10,000. For scans in $N_{\rm dim} = 7$ or 15 dimensions, we test `nTrialLists` values of $N_{\rm dim}$, $N_{\rm dim}+1$ and $N_{\rm dim}+2$. For the two-dimensional scans, we consider a larger range, setting `nTrialLists` to 2, 4, 24 and 48. We plot a selection of these results in Fig. 13.

In two dimensions, we see that more chains result in some improvement in the reliability of the algorithm in uncovering competitive values of the best-fit likelihood. Unsurprisingly,

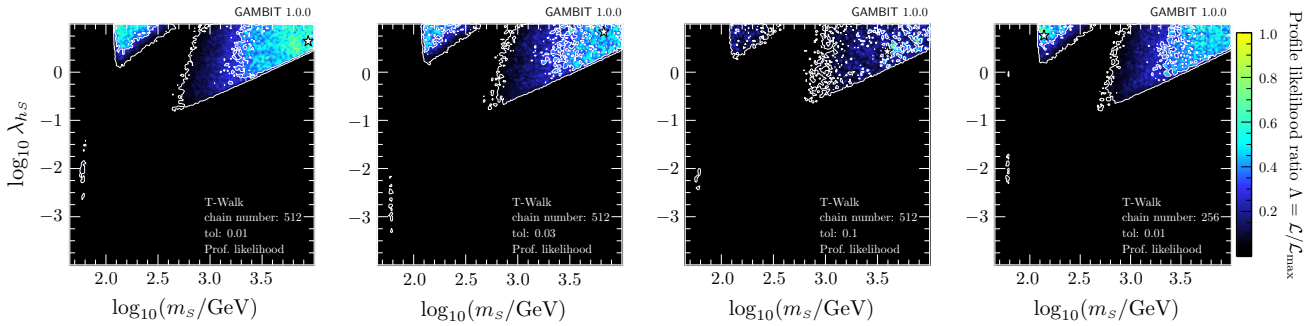


Fig. 11 Profile likelihood ratio maps from a 15-dimensional scan of the scalar singlet parameter space, using the T-Walk scanner with various numbers of chains and different tolerances. The maximum likelihood point is shown by a white star

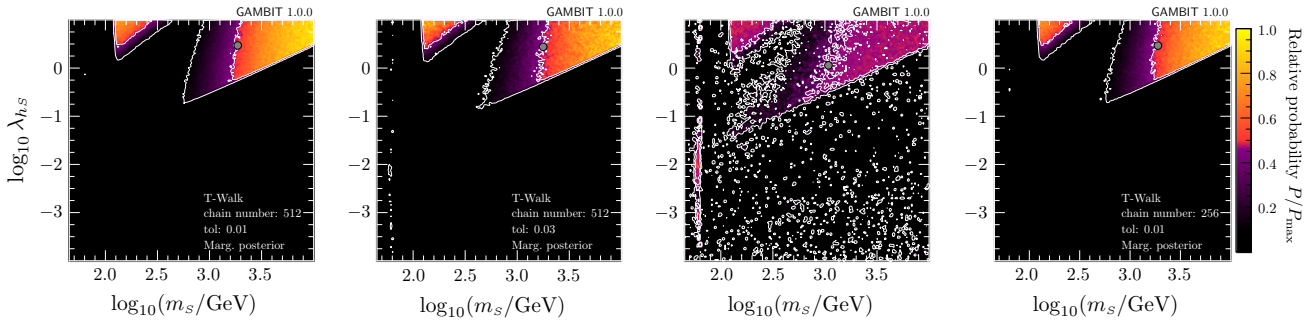


Fig. 12 Marginalised posterior probability density maps from a 15-dimensional scan of the scalar singlet parameter space, using the T-Walk scanner with various numbers of chains and different tolerances. The second to rightmost panel is from a 512-chain scan with a tolerance of 0.1. Note that the colourbar strictly only applies to the rightmost panel, and that colours map to the same enclosed posterior mass on each plot, rather than to the same iso-posterior density level (i.e. the transition from red to purple is designed to occur at the edge of the 1σ credible region, and so on). The posterior mean is shown with a grey bullet point

Fig. 13 also illustrates a tendency for longer chains to uncover slightly better fits. These trends are both borne out substantially more strongly in seven and fifteen dimensions. Visual inspection of the profile likelihood maps in Fig. 14 indicates that beyond `nTrials` of about 1000, these improvements in best-fit likelihood with increasing numbers of chains do not come with any substantial impact on the overall quality of sampling across the rest of the parameter space. We do notice a small runtime improvement, however. For example, two two-dimensional scans, each with 10,000 samples per chain, took 119 min to complete with `nTrialLists` = 48, but 165 min with `nTrialsLists` = 4. The best-fit log-likelihoods returned by the two scans were equal to the third significant figure. This timing difference reflects the improvement in acceptance that can be achieved when GreAT is able to draw on many different chains for constructing its correlation matrix.

In Fig. 15, we show the posterior maps resulting from the final set of independent samples returned by GreAT after its thinning process. Clearly, none of the scans we have run produce enough independent samples for a convergent map of the posterior, at least at the relatively high bin resolution that we employ for these tests.

For all scans, we observe that a minimum value between 1000 and 10,000 for `nTrials` is required in order to achieve a consistent value for the best-fit log-likelihood. We also notice that very low values (below ∼1000) map the profile likelihood rather poorly. The value of `nTrialLists` appears to be less crucial to the quality of the result; in general, values of $N_{\text{dim}} + 1$ and above appear to give relatively stable results when coupled with `nTrials` $\gtrsim$ 10,000. Substantially longer chains (`nTrials` $\gg$ 10,000) would probably be required to obtain high-resolution posterior maps.

11.5 The effect of dimensionality on performance

We have studied scanner performance in detail for two, seven and fifteen-dimensional parameter spaces, by increasing the number of nuisance parameters; each additional parameter adds an additional Gaussian component to the likelihood, and modifies the existing components. We now fix the computing configuration and scanner parameters (or apply a consistent scaling with dimensionality, where appropriate), and carry out scans for every possible dimensionality from two to fifteen. The results of these tests are presented in Fig. 16. The scanner settings we use for these tests are:

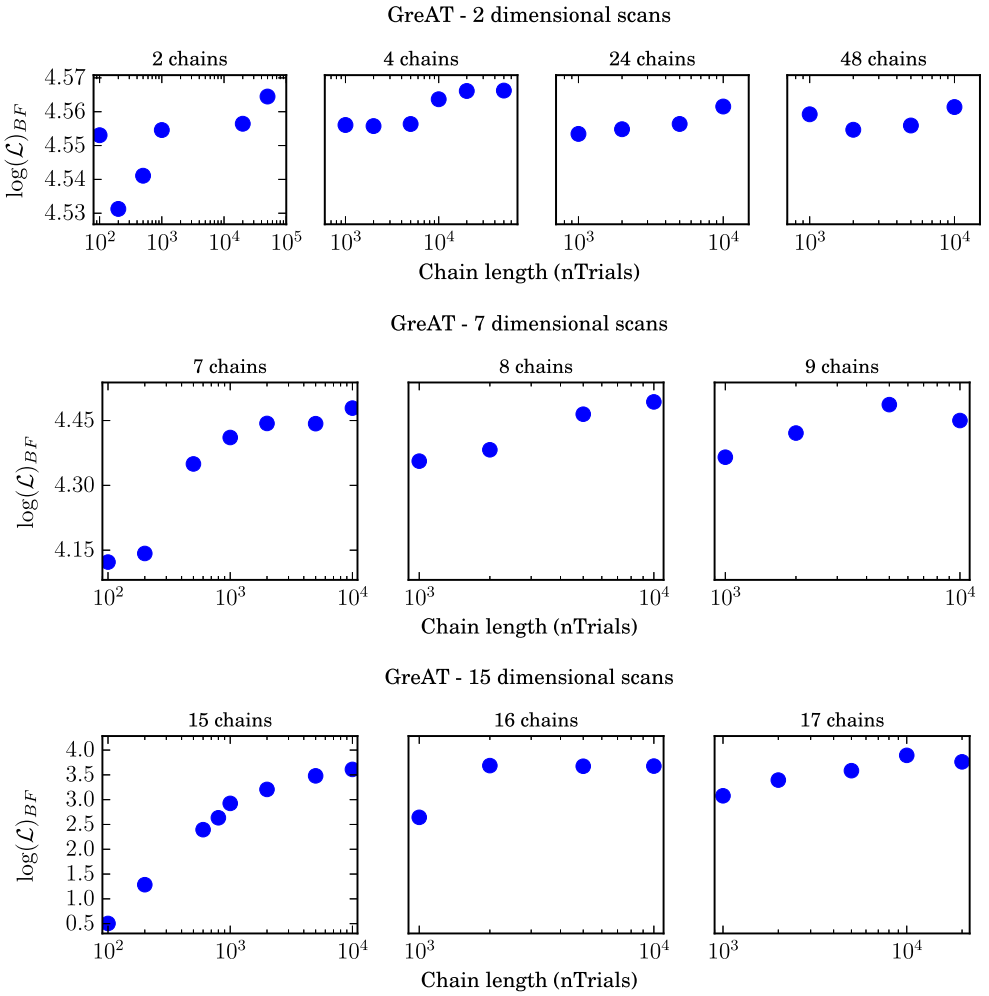


Fig. 13 Best-fit log-likelihoods for scans using the GreAT sampler in two (top row), seven (middle row) and fifteen dimensions (bottom row). The number of chains is set by the `nTrialLists` parameter. Note that the likelihood is dimensionful, leading to $\mathcal{L}_{\rm BF} > 1$ [29]

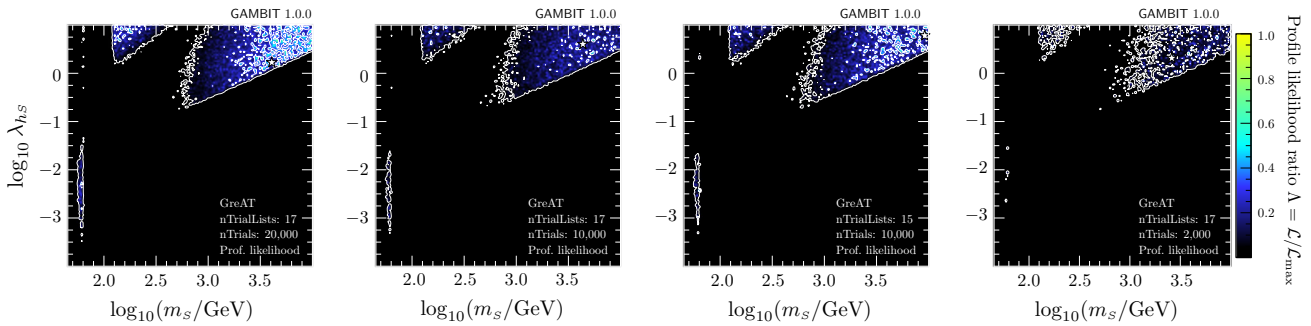


Fig. 14 Profile likelihood ratio maps from a 15-dimensional scan of the scalar singlet parameter space, using the GreAT sampler with various numbers of chains (`nTrialLists`) and chain lengths (`nTrials`). The maximum likelihood point is shown by a white star

Diver: `NP` = 20,000, `convthresh` = 10^{-3}
MultiNest: `nlive` = 20,000, `tol` = 10^{-3}
T-Walk: `chain_number` = number of MPI processes + $N_{\rm dim} + 1$, tol = `sqrtR` − 1 = 0.05
GreAT: `nTrials` = 2000, `nTrialsList` = $N_{\rm dim} + 1$

To reach convergence, GreAT requires significantly more likelihood evaluations for a larger number of dimensions. Although this is undoubtedly in part due to the increased number of chains used in higher dimensions, even with this increased number of evaluations, the best-fit log-likelihood is not competitive with that achieved by either Diver or Multi-

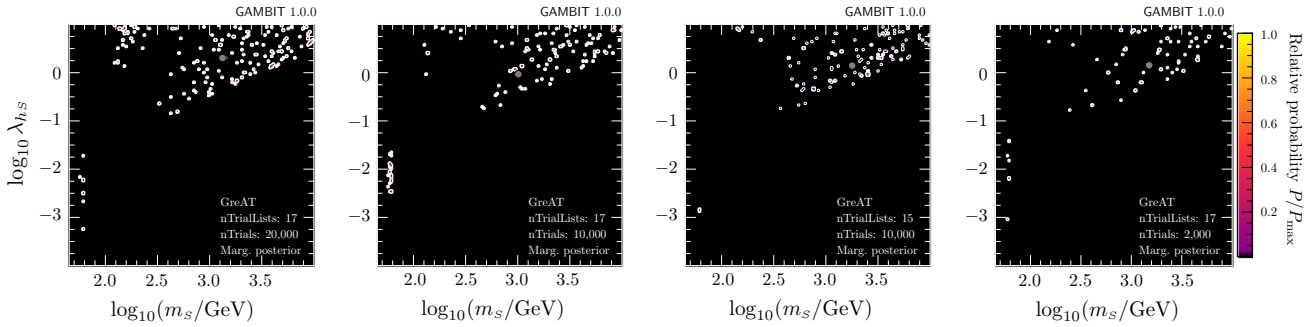


Fig. 15 Marginalised posterior ratio maps from a 15-dimensional scan of the scalar singlet parameter space, using the GreAT sampler with various numbers of chains (`nTrialLists`) and chain lengths (`nTrials`). Note that the colourbar strictly only applies to the rightmost panel, and that colours map to the same enclosed posterior mass on each plot, rather than to the same iso-posterior density level (i.e. the transition from red to purple is designed to occur at the edge of the 1σ credible region, and so on). The posterior mean is shown with a grey bullet point

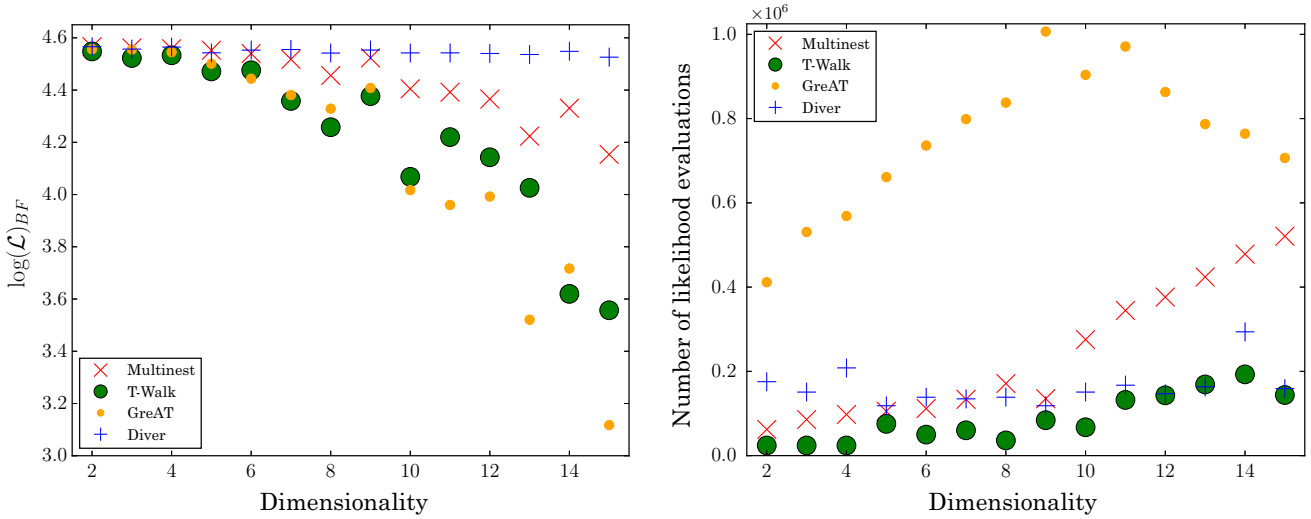


Fig. 16 Best-fit log-likelihood (left) and number of likelihood evaluations (right) as a function of dimensionality, for all four scanning algorithms, using a fixed computing configuration and scanner settings. Note that the likelihood is dimensionful, leading to $\mathcal{L}_{\mathrm{BF}} > 1$ [29]

Nest. If we demanded that all scanners must achieve the same quality of best fit, then it is clear that GreAT would require an even greater number of function evaluations to achieve this. Judging from the quality of best fit, the decrease in the number of evaluations required for convergence by GreAT in higher dimensions is clearly the result of spurious early convergence, rather than any increase in performance.

Diver performs extremely well at all dimensionalities, out-performing the other three scanners in terms of quality of best fit at $N_{\mathrm{dim}} \geq 10$. It also achieves this using a consistent number of likelihood evaluations across the full dimensionality range. MultiNest is able to achieve a competitive best-fit log-likelihood up until $N_{\mathrm{dim}} \sim 10$, however this comes with a steady increase in the number of evaluations with respect to dimensionality. T-Walk runs for a consistent number of likelihood evaluations across all dimensions, despite the required increase in number of chains, yet the best-fit deteriorates significantly with respect to dimensionality, in much the same way as it does with GreAT. The ensemble version of the MCMC algorithm implemented by T-Walk essentially provides the same best-fit performance as the regular MCMC (GreAT), but with a significant improvement in efficiency with increasing dimension. Overall, at least in this parameter space, Diver appears to be the scanner of choice for larger dimensions.

11.6 Scanning efficiency

The number of likelihood evaluations required to reach convergence is not the only reasonable metric for scanner efficiency. In general the number of evaluations is used as a proxy for time, as the likelihood evaluations are generally expected to be the bottleneck in most scans – but it is also illustrative to look directly at actual runtime. The efficiency of a scanner

can be degraded by poor use of parallel processing capabilities, or by complicated calculations performed between likelihood evaluations. This can lead to a divergence between the apparent performance assessed purely by number of function evaluations, and the true walltime needed. We therefore record the actual CPU time used for all scans, and compare with the total number of likelihood evaluations in Fig. 17.[4]

Figure 17 shows that dimensionality has a significant impact on the relative efficiency per likelihood evaluation of each algorithm. For two-dimensional scans, we see that T-Walk performs the least efficiently, while the other algorithms are reasonably similar. However, in the higher-dimensional parameter spaces, the efficiency of the nested sampling in MultiNest becomes comparable to the MCMC in T-Walk, whereas GreAT and Diver remain relatively efficient. The reduction in performance by MultiNest in higher dimensions is probably due to the complicated calculations required to perform its ellipsoidal sampling of multi-dimensional modes. These calculations must be performed between each generation of live points. Another potential cause of the performance reduction in T-Walk and MultiNest is the intrinsic level of parallelisability of their algorithms, relative to the other scanners. For problems with larger numbers of parameters, we observe that the most efficient sampling algorithms are GreAT and Diver, with both exhibiting the lowest average latency between likelihood evaluations.

In Fig. 18, we summarise the overall performance of the algorithms in terms of time and fit quality at each dimensionality. We bin all completed test scans logarithmically in the total convergence time, and for each sampler, choose the scan in each bin with the best fit. There are no Diver points in the longer bins, simply because the longest Diver scans took less time than the longest scans with other samplers. Diver clearly outperforms the other algorithms in high dimensions by this metric as well, finding a better fit in a shorter runtime than the other three algorithms. It is also important to note the vertical scales in Fig. 18, where the likelihood values span a much wider range in seven and fifteen dimensions than in two. On close inspection however, we can see even in two dimensions that Diver and MultiNest obtain better fits in less time than either T-Walk or GreAT.

We also notice that in higher dimensions, although T-Walk takes less evaluations than GreAT, both take a similar amount of runtime to reach convergence, suggesting that T-Walk's reduced sampling is offset by additional algorithmic complexity requiring more extended calculations *between* samples.

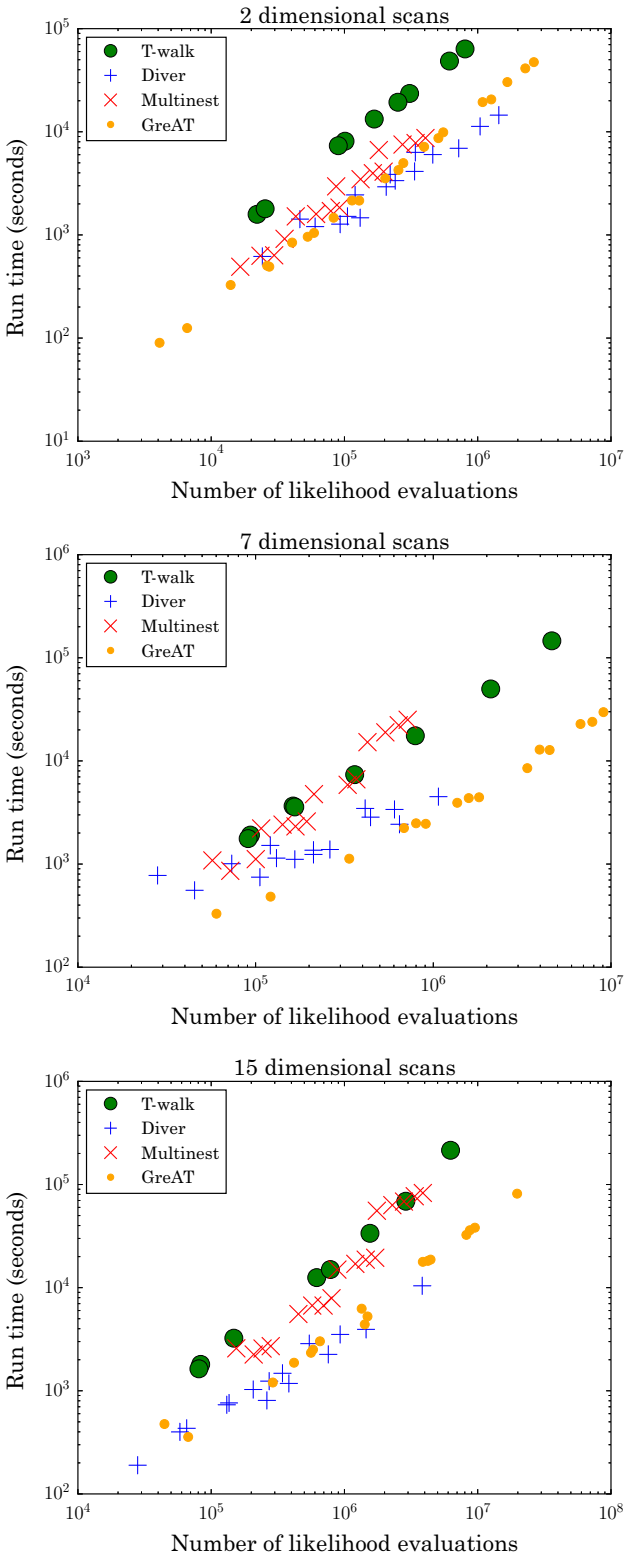


Fig. 17 The real time required as a function of likelihood evaluations for two- (upper), seven- (middle) and fifteen-dimensional (lower) scans

[4] Here we use 24 processes for the two dimensional scans, and 240 processes for the seven and fifteen-dimensional scans, so time comparisons should not be drawn between the two-dimensional plots and the seven/fifteen-dimensional ones.

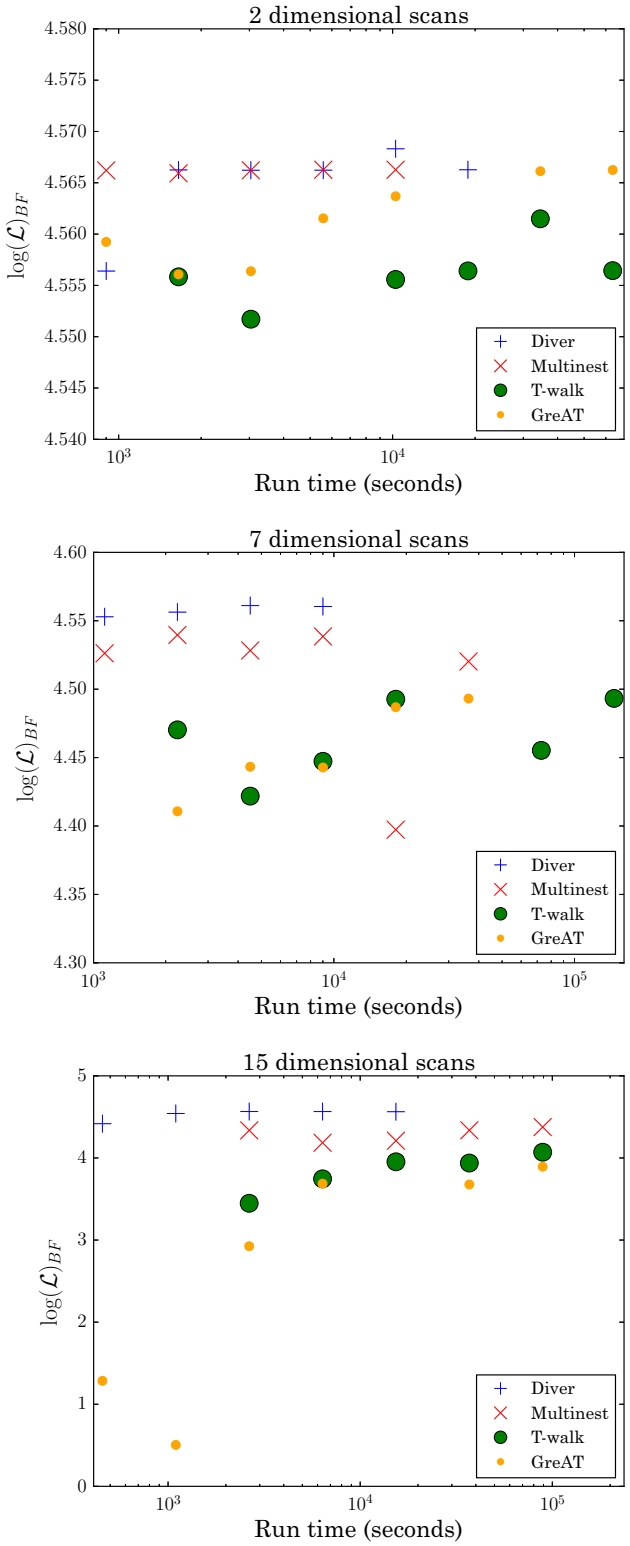


Fig. 18 The best-fit likelihood achieved by each scanner within a given time limit, for two (upper), seven (middle) and fifteen-dimensional (lower) scans

11.7 Posterior sampling

Figures 8, 12 and 15 show the posterior sampling abilities of T-Walk, MultiNest and GreAT, respectively. The best-quality posterior in T-Walk took 9 hr, while in MultiNest the best posterior we show took over 21 hr. The highest-quality GreAT posterior we show took even longer, and is clearly a poorer result than what was achieved by T-Walk and MultiNest.

Comparing the quality of the posterior maps achieved by T-Walk and MultiNest reveals some interesting trends. Firstly, despite taking less than half the runtime, the best posterior map returned by T-Walk appears to have given a better-converged map of the posterior than the best effort by MultiNest.

We can also see a distinct tendency for the shapes of the contours returned by MultiNest to erroneously 'smooth away' sharper features in the posterior, which are mapped far more carefully and accurately by T-Walk. This is most likely due to the ellipsoidal sampling method intrinsic to MultiNest, which biases the algorithm towards finding new live points within elliptically-shaped regions encompasing its current population of points. This makes it rather easy for the algorithm to miss sharp features in the posterior, such as the low-coupling tip of the highest-mass mode in the scaler singlet parameter space, which would protrude beyond the approximate contour defined by the bounding ellipsoids in MultiNest.

We also see that posterior maps become poorer for shorter scans with both T-Walk and MultiNest, but in quite distinct ways. In MultiNest, a scan performed with too few live points or too high a tolerance will give a poorly-sampled posterior with few favoured regions, essentially because the algorithm has only managed to locate the most dominant modes of the posterior at the outset. In contrast, a poorly-converged T-Walk scan, particularly one with a large `tol` value, will typically instead result in a map that includes all relevant modes across the parameter space, but with their relative contributions poorly determined, such that they appear alongside a number of other, spurious, favoured regions. When inspecting a posterior map, particularly from brief scans, it is important to be aware of these differences between the algorithms.

11.8 Discussion

We have investigated the performance of the four major samplers available in ScannerBit as part of GAMBIT 1.0.0, over a range of algorithmic settings and parameter space dimensionalities. In Table 2, we summarise our recommended values for the two most important settings of each scanner. These are intended as starting values that will give reasonably robust results. However, every parameter space is different and a publication quality results may require significantly more

Table 2 The recommended starting parameters for each scanner available in GAMBIT 1.0.0. Here N_{dim} is the dimensionality of the scan and N_{MPI} is the number of (distributed-memory) parallel processes available to GAMBIT

Scanner	Parameter	Recommendation
MultiNest	`nlive`	2×10^4
	`tol`	10^{-3}
Diver	`NP`	2×10^4
	`convthresh`	10^{-3}
T-Walk	`chain_number`	$N_{\text{dim}} + N_{\text{MPI}} + 1$
	`sqrtR`	< 1.01
GreAT	`nTrialLists`	$N_{\text{dim}} + 1$
	`nTrials`	10^4

stringent settings, in order for final results to be sufficiently robust. See Sects. 11.2–11.4 for more detailed recommendations.

We are also able to make detailed comparisons between the four scanning algorithms. In Sects. 11.5 and 11.6 it became evident that differential evolution, as implemented in Diver, consistently out-performs the other algorithms in the computation of profile likelihoods. This becomes particularly clear in high dimensions, where Diver leads the other algorithms in likelihood mapping, the quality of the best fit found, and overall efficiency.

The true best-fit point for this likelihood is located in the low-mass region, regardless of the number of additional free parameters. The scanners did not always locate this point, and in many cases located a best-fit in one of the high-mass modes. Although locating this point in two dimensions is less challenging (see Appendix E), once the dimensionality is increased, only Diver (with most stringent convergence criteria) was able to successfully locate the best fit in the low-mass mode. All other scans converged to a best fit in a completely different mode, demonstrating the value of using alternative algorithms to fully understand the parameter space.

For careful mapping of the posterior, we find that T-Walk is the most effective algorithm, followed by MultiNest and GreAT. T-Walk manages to sample the posterior distribution at higher resolution in less time than the other two scanners, and avoids the ellipsoidal biases that appear to afflict MultiNest. For computing low-resolution posteriors however, MultiNest has the advantage that it requires less parameter tuning than T-Walk, and can more quickly identify which are the most relevant posterior modes.

In many cases, having both Bayesian and frequentist interpretations of results is desirable. This makes it necessary to use a sampler able to effectively sample the posterior, such as MultiNest or T-Walk. However, our tests show that this is best performed *after* the likelihood function has been carefully mapped with another sampler, in order to find all modes. For example, in Fig. 7, MultiNest has completely missed the likelihood mode at low mass. This mode was successfully found by all three of the other samplers. If MultiNest were to be used exclusively, then this region – which contains best-fit points degenerate with those in the other modes – would be completely unexplored. However, with the knowledge gained from the other scanners, a localised study can be performed using MultiNest around the low-mass region (a technique used in Refs. [33,35]), in order to correctly evaluate the full posterior. In this way, the ability to use complementary scanners significantly improves the statistical robustness of results.

For lower-dimensional problems where both posterior distribution and profile likelihood are required, MultiNest could potentially be used solo, to save repeating analyses with multiple scanners. We find that it is able to locate all modes when scanning only the two-dimensional parameter space, and that it is reasonably efficient compared with the other algorithms. In general though, relying on only a single sampling algorithm is risky.

The two MCMC-based scanners available in GAMBIT 1.0.0, T-Walk and GreAT, provide the user with a somewhat more traditional class of sampling methods. Although these algorithms are demonstrably less effective scanners in higher-dimensional profile likelihood problems, they may suit lower-dimensional studies better.

Notably, our tests here are based on only one physical problem; although this is intended as a realistic example, no single example could ever represent the full diversity of problems that might be encountered. Other parameter spaces and likelihood functions may therefore reveal different trends to those we have observed with the scalar singlet model.

12 Conclusions

In this paper we have presented ScannerBit, the statistical and sampling module for the new global fitting package GAMBIT. ScannerBit manages the overhead associated with choosing parameter combinations and applying prior transforms, and offers an extremely flexible framework into which any existing sampling code can be easily integrated. It is able to perform sampling in standard random, grid and raster patterns, or employ more sophisticated statistical methods including nested sampling, differential evolution, Markov Chain Monte Carlo and ensemble Monte Carlo. It interfaces seamlessly with the GAMBIT printer system to allow statistical and physical outputs of parameter scans to be saved to a common format of choice, entirely independent of the model under investigation or the sampling algorithm in use. It can also post-process existing sets of samples previously computed and saved with GAMBIT. ScannerBit can be used from within GAMBIT, or as a standalone package

independent of GAMBIT, allowing the user to connect to an arbitrary likelihood function and sample it using their desired algorithm.

In addition to ScannerBit itself, we have presented a new standalone sampling package based on differential evolution: Diver. Diver features a full suite of differential evolution variants, from standard rand/1/bin to adaptive and discrete versions, and additional operation modes designed to provide approximate Bayesian results. We have also presented a new implementation of the T-Walk algorithm, implemented natively in ScannerBit.

We compared the performance of the four main sampling algorithms interfaced to ScannerBit in GAMBIT 1.0.0: Diver, MultiNest, T-Walk and GreAT. We found that for profile likelihood analyis at low dimensionality, Diver and MultiNest outperform T-Walk and GreAT, and provide roughly equivalent performance to each other. At higher dimensions (10 and above), Diver substantially outperforms the other three algorithms on all metrics. T-Walk provides a more accurate, timely and complete mapping of the Bayesian posterior than MultiNest, although MultiNest identifies the primary posterior mode more quickly.

ScannerBit and GAMBIT can be obtained from gambit.hepforge.org, and are both released under the terms of the standard 3-clause BSD license.[5] Diver can be downloaded from diver.hepforge.org, or installed automatically from within GAMBIT by simply typing `make diver`; it is released under a license that makes it free to use and distribute for academic and non-profit purposes.

Acknowledgements We thank the other members of the GAMBIT Collaboration for helpful discussions. We warmly thank the Casa Matemáticas Oaxaca, affiliated with the Banff International Research Station, for hospitality whilst part of this work was completed, and the staff at Cyfronet, for their always helpful supercomputing support. GAMBIT has been supported by STFC (UK; ST/K00414X/1, ST/P000762/1), the Royal Society (UK; UF110191), Glasgow University (UK; Leadership Fellowship), the Research Council of Norway (FRIPRO 230546/F20), NOTUR (Norway; NN9284K), the Knut and Alice Wallenberg Foundation (Sweden; Wallenberg Academy Fellowship), the Swedish Research Council (621-2014-5772), the Australian Research Council (CE110001004, FT130100018, FT140100244, FT160100274), The University of Sydney (Australia; IRCA-G162448), PLGrid Infrastructure (Poland), Polish National Science Center (Sonata UMO-2015/17/D/ST2/03532), the Swiss National Science Foundation (PP00P2-144674), the European Commission Horizon 2020 Marie Skłodowska-Curie actions (H2020-MSCA-RISE-2015-691164), the ERA-CAN+ Twinning Program (EU & Canada), the Netherlands Organisation for Scientific Research (NWO-Vidi 680-47-532), the National Science Foundation (USA; DGE-1339067), the FRQNT (Québec) and NSERC/The Canadian Tri-Agencies Research Councils (BPDF-424460-2012).

Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.
Funded by SCOAP[3].

Appendix A: Sources, options and outputs of the Diver package

A.1: Sources

Each of the source files located in `diver/src/` contains a single eponymous Fortran module:

`de.f90`: the main module of Diver, containing the function `diver()`, by which the package is invoked.
`init.f90`: contains routines to set all parameters for the run and to initialise the population every generation
`mutation.f90`: contains routines to allow standard DE mutation following Eq. 23 and self-adaptive mutation using jDE or λjDE (see Sect. 10.2.2).
`crossover.f90`: contains routines to allow binomial or exponential crossover, or self-adaptive crossover using jDE or λjDE.
`selection.f90`: performs selection of the next generation of vectors, applies boundary conditions, and removes duplicate vectors to ensure population diversity (see Sect. 10.2.4). If MPI is used, this is where most MPI routines are called.
`converge.f90`: checks whether the population has converged sufficiently to end the current DE run.
`io.f90`: saves the parameters of the run as well as the population at regular intervals. Contains routines to continue a run that was stopped partway through.
`evidence.f90`: contains routines used for calculating approximate Bayesian evidence values.
`posterior.f90`: contains routines used for calculating approximate Bayesian posterior probability density functions.
`detypes.f90`: contains interfaces to the likelihood function and prior, as well as the definitions of the internal data types used by Diver.
`deutils.f90`: contains utility routines.
`cwrapper.f90`: acts as an interface between C/C++ drivers and `de.f90`.

[5] http://opensource.org/licenses/BSD-3-Clause. Note that fjcore [65] and some outputs of FlexibleSUSY [66] (incorporating routines from SOFTSUSY [67]) 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.

A.2: Run options

Options for a Diver run are passed directly as arguments to `diver()` or `cdiver()`. The required arguments are:

`double precision func()`: The function to optimise, assumed to be positive definite; should generally correspond to the negative log-likelihood for statistical scans. See example driving programs for suggested use. Must take the following arguments:

`double precision params`: An array of size equal to the sum of D (the dimensionality of the parameter space) and `nDerived`, the number of derived quantities to be output in the run.

`integer fcall`: The total number of calls to `func`; should be incremented appropriately by the objective function.

`logical quit`: A flag set by the objective function. If this is ever set to `true`, Diver will save and quit at the end of the current generation.

`logical validvector`: A flag set by Diver. If this is `false`, the point in parameter space represented by `params` is outside the specified parameter boundaries, and should not be evaluated.

`c_ptr context`: A context pointer, allowing the driving program to pass arbitrary information to `func`. Can be modified in a call to `func`, and will retain its value the next time the function is called.

`double precision lowerbounds`: An array of size D, giving the desired lower bounds of the parameter space.

`double precision upperbounds`: An array of size D, giving the desired upper bounds of the parameter space.

`character path`: The path to which output files should be saved.

Other arguments are optional and default to sensible values if left unspecified. Here we list these in the format *option*`[`*default*`]`:

`integer nDerived[0]`: The number of derived quantities to be calculated by the likelihood/objective function. If `outputSamples` is `true`, these are saved in human-readable format along with the original parameters in a `.sam` file.

`integer discrete[empty]`: A vector listing all dimensions of the parameter space that should be treated as discrete parameters. See Sect. 10.2.3 for details.

`logical partitionDiscrete[false]`: Evolve discrete parameters as separate populations. See Sect. 10.2.3 for details.

`integer maxciv[2000 if doBayesian else 1]`: The maximum number of 'civilisations' to run. A civilisation is a full DE run with multiple generations, which terminates either because it has converged or reached generation number `maxgen`. If `doBayesian` is `true`, Diver will run additional civilisations up to `maxciv` until the approximate Bayesian evidence has converged; if `doBayesian` is `false`, Diver will simply repeat DE optimisation `maxciv` times, and save the results as a single set of samples.

`integer maxgen[300]`: The maximum number of generations for the DE run. Usually the default convergence criterion will cause Diver to end the DE run before this number has been reached.

`integer NP[10*size(lowerbounds)]`: The population size. Larger populations take longer to run but are less likely to become trapped in local minima. Small populations run more quickly because they require fewer likelihood/objective evaluations per generation, but they lack diversity and may converge prematurely. The default is set to $10D$; we recommend that `NP` never be set to less than D. If Diver is invoked using MPI, the actual population size will be increased from the requested size until it is a multiple of the number of MPI processes to be used.

`double precision F[0.7]`: The mutation scale factor(s); see Sects. 10.1.1 and 10.1.4. This should be supplied as an array. The scale factor, and the degree to which the population is spread out, together determine the radius around the population in which new points can be proposed. For this reason, `F` should be smaller than 1, to help convergence, but not too small, to prevent premature convergence. This option is ignored when `jDE` or `lambdajDE` is `true`.

`double precision Cr[0.9]`: The crossover rate; see Sects. 10.1.2 and 10.1.4. This option encourages mixing between the trial and target vectors, and can encourage search along individual dimensions. This parameter should be set between 0 and 1, inclusive. If it is set to 0, trial vectors will differ from the target vector along only one dimension. If it is set to 1, trial vectors will be entirely unrelated to their target vectors. This option is ignored when `jDE` or `lambdajDE` is `true`.

`double precision lambda[0]`: A scale factor linking the best target vector in the population to the initial vector chosen for mutation; see Sect. 10.1.4. This may take any value between 0 and 1, inclusive. If `lambda = 0`, the best vector is not used for mutation. If `lambda = 1`, mutation will use the best vector as the starting point for all new vectors. As a result, setting `lambda` > 0 will cause DE to optimise more aggressively. This option is ignored when `jDE` or `lambdajDE` is `true`.

`logical current[false]`: Use the current target vector as a base for mutation; see Sect. 10.1.4. This option is ignored when `jDE` or `lambdajDE` is `true`.

`logical expon[false]`: Use exponential crossover instead of binomial; see Sect. 10.1.4. This option is ignored when `jDE` or `lambdajDE` is `true`.

`integer bndry[1]`: Controls the behaviour when trial vectors are outside the allowed boundaries of the parameter space. Should be set to an integer between 1 and 4:

- `1` ('brick wall'): points outside the boundaries are rejected during the selection phase.
- `2` ('random reinitialisation'): For each point outside the bounds, a random new valid point is chosen.
- `3` ('reflection'): points outside the boundaries are reflected across the limits so that they land inside. This option is recommended if full exploration of the edges of parameter space is desired.
- $\geq$`4` (none): boundary conditions are not enforced. This may lead to the population drifting away from the initially specified region of parameter space, and should be used with caution.

`logical jDE[true]`: Use self-adaptive rand/1/bin DE, as described in Sect. 10.2.2. If this option is `true`, the values set for `F`, `Cr`, `lambda`, `current`, and `expon` are ignored. This option is ignored when `lambdajDE` is `true`.

`logical lambdajDE[true]`: Use self-adaptive rand-to-best/1/bin DE, as described in Sect. 10.2.2. If this option is `true`, the values set for `F`, `Cr`, `lambda`, `current`, `expon`, and `jDE` are ignored. If less aggressive optimisation is required, we recommend that this be turned off, and jDE used instead.

`double precision convthresh[0.001]`: The threshold for convergence of one DE population (a 'civilisation'). The smoothed fractional improvement in the population over successive generations must drop below this value for a population to achieve convergence. Assuming that the likelihood/objective function (func()) has been chosen to return $\ln \mathcal{L}$, the smoothed fractional improvement in the mean is defined as

$$\delta_{\text{smooth}} = \frac{1}{n} \sum_{i=j}^{j-n+1} \left[1 - \frac{\sum_{\text{population}} \ln \mathcal{L}_{i-1}}{\sum_{\text{population}} \ln \mathcal{L}_i} \right], \qquad \text{(A.1)}$$

where i is the generation index, j is the current generation number, and n is the population smoothing length, given by `convsteps`.

`integer convsteps[10]`: The number of generations over which to smooth the fractional improvement of the mean population value of the likelihood/objective function when testing for convergence.

`logical removeDuplicates[see Sect. 10.2.4]`: Remove duplicate vectors within a single generation. Turning this on is generally good for population diversity. Duplicates are however exceedingly rare when either `jDE` or `current` is `true`, so keeping `removeDuplicates` = `true` in these cases is not necessary, but can be a useful debug check against MPI problems.

`logical doBayesian[false]`: Estimate posterior weights of population members, and the natural log of the Bayesian evidence $\ln Z$; see Sect. 10.2.5.

`double precision prior()`: The prior function to be accounted for in approximate Bayesian computations; see Sect. 10.2.5. Required if `doBayesian` is `true`, ignored otherwise.

`integer maxNodePop[1.9]`: The population above which to perform node division in the binary spanning tree used to estimate posterior weights; see Sect. 10.2.5. Ignored unless `doBayesian` is `true`.

`double precision Ztolerance[0.01]`: The fractional uncertainty in $\ln Z$ taken to indicate convergence of the evidence; Sect. 10.2.5. Ignored unless `doBayesian` is `true`.

`integer savecount[1]`: The number of generations that should pass between periodic saves of the population.

`logical resume[false]`: Resume from a previous run.

`logical outputSamples[true]`: Write samples and derived quantities in an output `.sam` file. Even if this is `false`, the `.sam` file will still be written if `discrete` is non-empty.

`integer init_population_strategy[0]`: Strategy to employ when initialising the first generation. Should be set to an integer between 0 and 2:

- `0` ('one-shot'): initialise each member of the first generation to a different random point drawn from between the stated `lowerbounds` and `upperbounds`, without regard to its fitness.
- `1` ('n-shot'): draw candidate initial population members randomly from between `lowerbounds` and `upperbounds`. Accept a candidate if its function value is below `max_acceptable_value`, otherwise attempt to draw an alternative candidate. Continue until `max_initialisation_attempts` is reached, then if a good candidate has still not been found, accept the next candidate without regard to its fitness.
- `2` ('fatal n-shot'): as per `1`, but throw a hard error if `max_initialisation_attempts` is reached when initialising any member of the first generation.

`integer max_initialisation_attempts[10000]`: Maximum number of times to try to find a valid vector when initialising each member of the initial population if `init_population_strategy` > `0`; ignored otherwise.

`double precision max_acceptable_value[`10^6`]`: The cutoff value of the objective function below which to consider a candidate initial population member 'accept-

able' if `init_population_strategy > 0`; ignored otherwise.

`c_ptr context[C_NULL_PTR]`: A raw `void` callback pointer, used to pass information from the driver program to the objective function. This is typically used to pass an external function address, which the objective function then uses to help with its evaluation.

`integer verbose[1]`: The amount of information to print to screen. Recognised values are:

- `0` ('Quiet'): Only error messages will be printed.
- `1` ('Laconic'): Prints warning messages and a summary at the beginning and end.
- `2` ('Chatty'): Prints civilisation-level and basic generation-level information.
- `3+` ('Verbose'): Prints detailed information for each generation.

A.3: Output formats

Diver produces up to four different output files, in plain ASCII format. The first three of these are always generated, and are needed for resuming a run.

path`.rparam`: the complete range of Diver settings in use in the current run, including optional parameters. The meaning of each entry in this file can be read off the comments provided in the routine `save_run_params` in `io.f90`. This file is created during the first save operation, which takes place after `savecount` generations have been completed (see Sect. A.2).

path`.devo`: convergence and other dynamic runtime information. This is the file to check for evaluating the progress of a given run. Its contents are as follows:

```
civilisation number, generation number
𝒵, ⟨P²⟩, Δ𝒵, unpolished 𝒵
N_s, individuals saved, number of calls to func

fitness at best fit θ_best
raw (non-discretised) parameter values at θ_best
parameter values and derived quantities at θ_best

fitnesses of current population
raw parameters of current population
parameters & derived quantities of current pop.

if jDE or lambdajDE:
  F values of current population
  Cr values of current population
  λ values of current population

δ_smooth
individual contributions to δ_smooth from each of
 the last convsteps generations
```

Further information can be found in the routine `save_state` of `io.f90`. Like the `.rparam` file, this file is created during the first save operation.

path`.raw`: the posterior weight, fitness, civilisation number, generation number and raw parameter values (in this order), for every individual so far generated in a scan. The data for each individual occupies a single line in the file. In order to allow proper resumption of the run, the sampled values of any discrete parameters appear as they are used internally for mutation, i.e. as values of a continuous parameter. This file is created before the initial population is generated.

path`.sam`: all parameter samples, in a similar format to the `.raw` file, but with additional columns for each derived quantity calculated in a scan. The sampled values of any discrete parameters are also given rounded to their true discrete values in this file, unlike in the `.raw` file. This file is only generated if `outputSamples = true` and either `discrete` is non-empty or `nDerived` $\neq$ `0`. This file is created immediately after the `.raw` file.

Appendix B: Scanner options and outputs

For quick reference, here we provide the ScannerBit YAML file options and output formats for all five of the major scanners mentioned in this paper: the postprocessor (Sect. 6), GreAT (Sect. 7), T-Walk (Sect. 8) MultiNest (Sect. 9) and Diver (Sect. 10).

B.1: Postprocessor

The YAML setup required to run the postprocessor spans two sections of the master YAML file: the usual `Scanner` section, plus also the `Parameters` section.[6] In the `Scanner` section, the options [and defaults] are as follows:

`like`: The purpose to use as the objective; should generally match the purpose set for likelihood components (e.g. in the `ObsLike` section of a GAMBIT YAML file).

`reweighted_like`: The output label used for the final result of `add_to_like` and `subtract_from_like` operations.

`add_to_like[empty]`: A vector of names of datasets present in the input samples, presumably log-likelihood values, to be added to the newly computed `like` and output as `reweighted_like`. (Note that the 'newly-computed' `like` may be zero if no entries in the GAMBIT `ObsLike` section have been assigned a `purpose` that matches `like`). For example, if the combined likelihood

[6] Some of the requirements of the `Parameters` section can be optionally implemented in the `Priors` section instead.

of a previous scan were labelled "LogLike", and one were to choose like:New_LogLike as the new composite (log-)likelihood for a new 'scan', then the way to ensure that the old and the new composite log-likelihoods were automatically summed for every model point would be to set add_to_like:[LogLike]. The results of this summation would appear in the new output with the label by reweighted_like.

subtract_from_like[empty]: As per add_to_like, except the old output is *subtracted* from like.

permit_discard_old_likes[false]: When set to false, this option forbids the purpose chosen for like from clashing with any data label in the input samples. For example: if the original purpose was LogLike, a different purpose must be chosen for like, or an error will be thrown. If this option is set true, then clashes are permitted, and will be resolved in the new output by replacing the old data with the newly-computed data (as occurs automatically for all other clashes between old and new dataset names). This option also applies to likelihood components listed in add_to_like, subtract_from_like, and reweighted_like. If set to false then these names may not be recomputed during postprocessing.

update_interval[1000]: Defines the number of iterations between messages reporting on the progress of the postprocessing.

reader: Options under this item specify the format of the old output file to be read, along with e.g. the path at which the file is located. The required options differ depending on which GAMBIT **printer** was used to save the results of the previous scan.

The final option, reader, is used to inform the postprocessor of the format and location of the old data that needs to be reprocessed. In this first release of GAMBIT there are only two possible **printer** formats, ascii and hdf5, as described in [29]. There are therefore at present only two sets of options that may be specified for the reader. For files created with the hdf5 printer:

type: hdf5

file: Path to the HDF5 file containing the data to be parsed

group: Group within the HDF5 file containing datasets to be parsed.

For ascii output:

type: ascii

data_filename: Path to the ASCII file containing the data to be parsed

info_filename: Path to the ASCII 'header' file that contains the labelling information for the columns of data_filename.

Note that the reader need not match the chosen **printer** in a postprocessing run; reading samples in ascii and outputting updated samples in hdf5, or vice versa, is permitted. This allows GAMBIT samples produced in one format to be easily converted into any other format.

Note also that where new print overloads have been defined for one or more printers, as described in Sect. 9.3 of Ref. [29], users wishing to postprocess the resulting data must *also* overload the equivalent _retrieve method of the reader in use, so that it can successfully read the new type in from the existing scan output. To do this, one needs to follow the instructions for adding a new print overload in Sect. 9.3 of Ref. [29], and then also add the body of the new _retrieve function to the file Printers/src/printers/*printer_name*/retrieve_overloads.cpp.

Using the postprocessor scanner also places some special requirements on the Parameters and/or Priors sections of the YAML file. First, the **models** chosen in the Parameters section must be a subset of the models that were used for the original scan. Secondly, the prior_type for all the parameters in those models must be set to none. This disables the standard GAMBIT prior system and allows the postprocessor to manually set parameter values (see Sect. 3.1.3 for details).

B.2: GreAT

The following options (with defaults in brackets) set the chain length and number of steps taken used by the GreAT sampler:

nTrialLists[10]: Number of Markov chains to be run.

nTrials[20000]: Number of steps in each Markov chain.

At the end of the run, the complete statistics for all chains run (burn-in length, correlation length, number of independent samples) are printed out in GreAT's native format. The independent samples and their multiplicities are stored and outputed to the GAMBIT printer system.

B.3: T-Walk

The options available for T-Walk in ScannerBit (with defaults in square brackets) are:

kwalk_ratio[0.9836]: ratio of walk and traverse to hop and blow moves. The default is to strongly prefer walk and traverse moves.

projection_dimension[4]: dimension of the projection subspace in which walk and traverse moves are performed.
walk_distance[2.5]: width of the distribution function for the distance of the walk move (a_w; see Eq. 11).
traverse_distance[6]: width of the distribution function for the distance of the traverse move (a_t, see Eq. 15).
gaussian_distance[2.4]: Gaussian jump parameter d for the hop and blow moves. See Eq. 19.
chain_number[1+projection_dimension+number of MPI processes]: total number of MCMC chains. T-Walk will be highly inefficient if this parameter is set to anything less than the default.
hyper_grid[true]: confines the search to the hypercube defined by the priors.
sqrtR[1.001]: the version of T-Walk in ScannerBit uses the Gelman-Rubin convergence diagnostic $\sqrt{R}$ [43] to determine when a scan has converged. This compares the inter-chain dispersion to the total dispersion of each parameter. Values closer to 1 are better converged; when $\sqrt{R}$ drops below the value given for sqrtR, the scan terminates.

The T-Walk scanner also outputs various variables associated with the scan to the GAMBIT printer system:

mult: Multiplicity (posterior weight) of each sample.
chains: Chain number for each sample. Rejected proposal points are assigned the number -1.

B.4: MultiNest

The ScannerBit plugin that runs the MultiNest sampler takes the following YAML options, which it passes directly through to the external MultiNest library (defaults are given in square brackets):

IS[true]: do nested importance sampling?
mmodal[true]: do mode separation?
ceff[false]: run in constant efficiency mode? Setting this true can result in poor evidence estimates.
nlive[1000]: number of live points.
efr[0.8]: required efficiency (only relevant if ceff = true).
tol[0.5]: stopping tolerance; the scan halts when the ratio of the estimated remaining unsampled evidence to the current estimate of the evidence drops below this value.
nClsPar[ndims]: number of parameters to do mode separation on. The default is to do separation on all parameters being scanned.
updInt[1000]: update interval; this sets the number of iterations between output file updates and any feedback passed to standard output. The MultiNest dumper function, which handles the calls to the GAMBIT printer, runs every 10*updInt iterations.
Ztol[-10^{90}]: the threshold in the logarithm of the evidence below which to ignore modes of the posterior.
maxModes[100]: expected maximum number of modes (used only for memory allocation).
seed[-1]: seed to use for the internal MultiNest random number generator. If this is negative, the seed is taken from the system clock.
fb[true]: provide feedback on run progress to standard output?
outfile[true]: write native MultiNest output files? ScannerBit does *not* add prior-transformed parameter values nor auxiliary observable values to the native MultiNest output, so this output is not very useful for analysis purposes. However, the native outputs are required for MultiNest to be able to resume scans that were previously interrupted. We recommend leaving this option set unless running scans that will definitely not need to be resumed.
maxiter[0]: maximum iterations permitted; a non-positive value is interpreted to mean infinity.

There are several other options that MultiNest ordinarily requires when run outside of ScannerBit, but for which ScannerBit can infer appropriate values and set automatically. These *cannot* be set in the Scanner section of the YAML file (although some can be changed indirectly by modifying the scan setup elsewhere):

Number of parameters (ndims): ScannerBit sets this option according to the number of varying parameters that exist in the model being scanned.
Size of 'cube' array (nPar): This is set to ndims+2. The first ndims slots contain the hypercube parameters, and in the extra two slots ScannerBit stores an ID number for each point, plus the MPI rank of the process that produced it. Together these two numbers uniquely identify every point sampled in a scan. These numbers are also stored in the GAMBIT printer system output, so they can be used to correlate the native MultiNest output with the GAMBIT printer output.
Resume mode (resume): ScannerBit activates resume mode by default unless the -r switch (for 'restart scan') is given at the command line.
Minimum loglike (logZero): points with $\ln \mathcal{L} <$ logZero will be ignored by MultiNest. This is set to 0.9999 times the value of model_invalid_for_lnlike_below in the likelihood node of the KeyValues section of the main YAML file.
Initialise MPI (initMPI): This is set to false because ScannerBit handles the initialisation of MPI.

Note that GAMBIT sets `logZero` to slightly more than `model_invalid_for_lnlike_below`. This is so that invalid points, assigned $\ln \mathcal{L}$ = `model_invalid_for_lnlike_elow` by the likelihood container [29], are treated as having zero likelihood by MultiNest. This is the desired behaviour during live point generation, as it prevents any of the initial live points being invalid.

During live point replacement however, this can prevent efficient parallelisation, as MultiNest requires all MPI nodes to continue testing proposed points until they each find one with $\ln \mathcal{L} >$ `logZero`. In complicated parameter spaces, where the ellipsoids encompass large regions of invalid parameter space, this can lead to many nodes idling whilst they wait for a small number of nodes to find their valid points, even if one of the points already found has a high enough likelihood to use for live point replacement. To circumvent this, following live point generation, when the MultiNest `dumper` function first runs, the MultiNest plugin communicates to ScannerBit and GAMBIT that likelihoods for invalid points should no longer be set to `model_invalid_for_lnlike_below`, but instead to the value of the alternative option `model_invalid_for_lnlike_below_alt`. This key can also found in the `likelihood` node of the `KeyValues` section of the main YAML file. The value of `model_invalid_for_lnlike_below_alt` defaults to half `model_invalid_for_lnlike_below`. Whenever it is set to more than `logZero` (i.e. 0.9999 times `model_invalid_for_lnlike_below`), MultiNest considers all samples found to be valid, and does not demand additional samples before evaluating whether those found are appropriate for live point replacement. We find that this often results in more than an order of magnitude improvement in performance when running MultiNest with $\mathcal{O}(100)$ or more MPI processes.

B.5: Diver

The YAML entry `KeyValues::likelihood::lnlike_offset` can be used to set the offset to be applied to the log-likelihood function passed to Diver, in order to maintain positive definiteness of the fitness function; this defaults to 10^{-4} times `KeyValues::likelihood::model_invalid_for_lnlike_below`.

The Diver interface in ScannerBit provides almost all of the run options mentioned in Sect. A.2, configurable directly from the Diver entry in the main YAML file. With a few exceptions, these options have the same names and default values as in Diver itself. The exceptions are:

- `NP` has no default, and must be specified in the YAML file
- `maxgen` defaults to `5000`, not `300`
- `bndry` defaults to `3`, not `1`
- `removeDuplicates` defaults to `true`, regardless of other options
- `outputSamples` is instead referred to by the YAML option `full_native_output`
- `init_population_strategy` defaults to `2`, not `0`
- `max_acceptable_value` defaults to 0.9999 times the value of `model_invalid_for_lnlike_below` in the `likelihood` node of the `KeyValues` section of the main YAML file
- `verbose` is instead referred to by the YAML option `verbosity`, and defaults to `0` instead of `1`.

Note that `doBayesian` is not available as a YAML option, and is hard-coded to `false`; there are multiple other scanners available in ScannerBit more efficient and accurate at scanning the Bayesian posterior than Diver. Correspondingly, `maxNodePop` and `Ztolerance` are not offered as YAML parameters either. Any user especially interested in obtaining posteriors from Diver running within ScannerBit should find this relatively easy to recode by comparison with e.g. the MultiNest or GreAT interface.

Appendix C: Custom priors

ScannerBit allows for users to add their own priors. These should be declared inside the `priors` namespace, in new headers placed in `ScannerBit/include/gambit/ScannerBit/priors`, and new source files placed in `ScannerBit/src/priors`.

Declaration of a new prior *prior_name*, arising from a new class *prior_class*, takes the form:

```
class prior_class : public BasePrior
{
  public:

    prior_class(const std::vector<std::string>&
     params, const Options& options)
    : BasePrior(params, cube_size)
    {//insert optional initialisation code}

    void transform(const std::vector<double>&
     unitpars, std::unordered_map<std::string,
     double>& outputMap) const
    {//insert non-optional transformation code}
};

LOAD_PRIOR(prior_name, prior_class)
```

Given this recipe, the only real input required of a user when implementing a new prior is to decide on its dimensionality (*cube_size*), and to write the body of its transformation function (*prior_class*`::transform`).

The class defining the user-specified prior inherits from the abstract base class `BasePrior`. This class has the following members:

`BasePrior(std::vector<std::string>, int)`: Base class constructor. Takes in a vector of strings that defines the parameter names, and an integer that specifies the dimension of the unit hypercube to be operated on by this prior. Typically this will be `1`, or the entire parameter space, available by simply calling the `size()` method on the vector passed as the first argument.

`void transform(std::vector<double>, std::unordered_map<std::string, double>)`: A pure virtual function that defines the prior transformation. Takes as input a vector of doubles with the input unit hypercube values, converts them to the actual model parameters, and stores them in the unordered map passed (by reference) as the second argument.

`unsigned int size(), unsigned int& sizeRef()`: Returns the dimension of the input unit hypercube.

`void setSize(int)`: Set the unit hypercube dimension.

`std::vector<std::string> param_names`: A protected member variable (i.e. accessible from derived classes only), which contains the names of the parameters as passed to the constructor.

A user-defined prior is registered in the ScannerBit prior database by invoking the following macro after the class declaration:

`LOAD_PRIOR(`*prior_name*`,` *prior_class*`)` Macro that loads the prior defined in class *prior_class*, and assigns it the internal name *prior_name*.

Here we give a worked example of the declaration of a custom prior. This prior is contained in the ScannerBit source file `ScannerBit/include/gambit/ScannerBit/priors/dummy.hpp`. This prior simply transforms the unit hypercube to the same unit hypercube.

```
namespace Gambit
{

  namespace Priors
  {

    class Dummy : public BasePrior
    {

      public:

        Dummy(const std::vector<std::string>&
         param, const Options&)
        : BasePrior(param, param.size())
        {}

        void transform(const std::vector<double>&
         unitpars, std::unordered_map<std::string,
         double>& outputMap) const
        {
          auto it_vec = unitpars.begin();
          for (auto it = param_names.begin(),
           end = param_names.end(); it != end;
           it++)
          {
            outputMap[*it] = *(it_vec++);
          }
        }

    };

    LOAD_PRIOR(dummy, Dummy)
  }

}
```

Here, the `Dummy` class inherits from the `BasePrior` class. The constructor passes the entered parameter names to the `BasePrior` constructor, as well as the hypercube size. The `transform` function transforms a `vector<double>` representing the unit hypercube into actual parameter values, which are stored in the output map. In this case, the hypercube values are directly stored in the output map. Lastly, the `Dummy` prior is loaded into the prior system and given the name `dummy`, by calling the macro `LOAD_PRIOR(dummy, Dummy)`.

Appendix D: Plugin Declaration and Interface

In the following subsections, we go through the definition, design, and operation of plugins in detail, starting with their declaration in Sect. D.1. ScannerBit provides a broad suite of utility functions that can be called from plugins. We first deal with the functions available to all plugins, for accessing information in the initialisation file of a scan (Appendix D.2), the chosen prior transformation (Appendix D.3), and the GAMBIT printers (Appendix D.4). We then list utility functions available only to scanner (Appendix D.5) or objective (Appendix D.6) plugins.

D.1: Plugin declaration

Source code for a plugin *plugin_name* is located within a directory `ScannerBit/src/`*plugins_kind*`/`*plugin_name*. Headers are found in `ScannerBit/include/gambit/ScannerBit/`*plugins_kind*`/`*plugin_name*. Here *plugins_kind* is either `scanners` or `objectives`.

Code for all plugins follows the same basic layout (with *plugin_kind* either `scanner` or `objective`):

```
#include "plugin_kind_plugin.hpp"

plugin_kind_plugin(plugin_name, version(...))
{
  environmental_macros

  plugin_constructor {...}

  return_type plugin_main(args) {...}

  plugin_deconstructor {...}
}
```

The plugin body can contain three blocks of code: a `plugin_constructor`, a `plugin_main`, and a `plugin_destructor`. The utility functions detailed in the following subsections can be accessed from within any of these three blocks. The `plugin_constructor` and `plugin_deconstructor` blocks will run when the plugin is loaded and unloaded, respectively. The code here can used to initialise, allocate, or deallocate variables needed by the plugin. The `plugin_main` block defines the function that will be run by the plugin. The form of the arguments for `plugin_main` required by ScannerBit depends on whether the plugin is a **scanner plugin** or an objective **(test function) plugin**.

For scanner plugins, `plugin_main` must take the form

```
void plugin_main() { code }
```

where *code* is the code that actually drives a statistical sampling algorithm. We give a full example of a minimal scanner plugin in Appendix D.5.1.

Objective plugins can be further categorised into 'likelihood' plugins, which compute likelihoods, and 'prior' plugins, which provide the transformation function needed to implement a ScannerBit prior (see Sect. 3). For likelihood plugins, `plugin_main` must be of the form

```
double plugin_main(const std::vector<double>&)
```

whereas for prior plugins, the required form is

```
void plugin_main(const std::vector<double>&,
 std::unordered_map<std::string, double>&)
```

We give a worked example of a minimal likelihood-oriented objective plugin in Appendix D.6.1.

Each plugin is built in a separate programming environment, with its own user-specified library dependencies and compile-time options. A set of *environmental_macros* that define the compilation environment can be declared at the beginning of a plugin. These macros can be used to define additional compilation flags, required libraries, required headers, or required entries in the input YAML file of a scan. The following macros are available:

`reqd_inifile_entries("X","Y",...)`: Indicates that the plugin will not be permitted to load unless the YAML node corresponding to the plugin in question, in the YAML input file of the scan, contains the options *X* and *Y*. Any number of required entries can be given as a comma-separated list.

`cxx_flags(flag_string)`: Additional flags to append to the compilation commands for this plugin.

`reqd_libraries("A","B",...)`: Tells ScannerBit to search for and link the libraries *A* and *B* if using this plugin. Any number of libraries can be given as a comma-separated list.

`reqd_headers("C","D",...)`: Specifies that the headers *C* and *D* must exist for the plugin to compile; any number of headers can be given in a comma-separated list. Like libraries, ScannerBit will automatically search for the specified headers.

If a library or a header listed in `reqd_libraries` or `reqd_headers` is in a non-standard location, or if ScannerBit is unable to locate it, the location can be specified in the `config/scanner_locations.yaml` or `config/objective_locations.yaml` configuration files.[7] Entries in the configuration files follow the format

```
plugin_name:
  plugin_version:
    - inc: include_dir
    - lib: library_path
```

This entry gives the locations of the libraries and headers needed for version *plugin_version* of the plugin *plugin_name*. Note that libraries require full paths, whereas headers require only an include directory. The plugin version can be given as "`any_version`", in which case the indicated library and/or header locations will be applied to every version of the plugin. If the `config/scanner_locations.yaml` or `config/objective_locations.yaml` configuration files do not exist, or a relevant entry is missing from them for a given plugin, then ScannerBit will use any relevant entry it can find in the files `config/scanner_locations.yaml.default` and `config/objective_locations.yaml.default`. These `.default` files ship with ScannerBit and should not be modified; it is up to the user to create `config/scanner_locations.yaml` and/or `config/objective_locations.yaml` if they wish to override or add to any of the defaults.

[7] Note that the current version of ScannerBit locates both libraries and headers at cmake time, not at runtime. This means that cmake must be run (or re-run) and ScannerBit rebuilt after scanners are built or moved. This is in contrast to the GAMBIT backend system, which locates and loads backend libraries entirely at runtime. It is expected that future versions of ScannerBit will dynamically load the shared plugin libraries, in line with GAMBIT backend practice.

D.2: Interface to input file

Detailed instructions on how to construct and format a YAML input file for a scan are given in Sect. 4. To extract entries from this file, the following functions are provided to both scanner and objective plugins:

ret_type `get_inifile_value<`*ret_type*`>(std::string key,` *ret_type* `default_value)`: Retrieves the value assigned to the YAML key `key`. If `key` is not present in the relevant part of the YAML file, an optional `default_value` to be returned can be specified. If no default is given, and the `key` is absent from the YAML file, ScannerBit will throw an error. The return value obtained will be interpreted as a quantity of type *ret_type*. Note that the key `default_output_path` will always return a value; if this key is not set in the YAML file, the output defaults to `scanner_plugins/`*plugin_name* (where *plugin_name* is the name of the plugin calling `get_inifile_value`). This is true for both scanner and objective plugins, although only scanner plugins are typically expected to generate output files.

`YAML::Node get_inifile_node(std::string key)`: Retrieves an entire YAML node with a given *key* from the input YAML file.

D.3: Interface to prior object

Both scanner and objective plugins can directly access the prior transformation object used in any given scan, via the function `get_prior()`. See Appendix C for details of how to use this object.

D.4: Interface to GAMBIT printer system

Within the body of a ScannerBit plugin, the `get_printer()` function can be called to obtain an object that acts as an interface to the GAMBIT the **printer** system. GAMBIT's printer system removes the need for scanners or their plugins to directly output sampled parameter values, as this responsibility is taken on by ScannerBit itself. The printer system also removes any need for scanners to output total likelihoods, individual likelihood components or observables; these are to be printed by objective plugins themselves, or in the case of GAMBIT, by the **likelihood container** (which is in effect just a very sophisticated likelihood plugin). This arrangement is designed to increase modularity, by allowing individual likelihoods to print their own – potentially highly model-specific – results, without the need to modify any scanner or scanner plugin code. Printing of scanner-specific quantities (such as posterior weights or chain multiplicities) must be handled by the scanner plugins themselves, and these quantities must be uniquely associated with specific parameter combinations. This is accomplished by assigning each parameter combination a unique point ID number via which the printer can associate any future outputs with a specific parameter combination.

The basic interface is contained within the `printer_interface` object returned by `get_printer()`. This object offers the following useful member functions:

`printer* get_stream(std::string name)`: Gets a pointer to the printer stream `name`. If no name is specified, the main printer is returned.[8]

`void new_stream(std::string name, YAML::Node option)`: Create a new printer stream named `name`, using the options contained in a YAML node `option` (which is itself optional). This typically only needs to be done on the MPI master process (See Ref. [29]). To then ensure that all MPI processes are aware of the new streams, the helper function `void assign_aux_numbers (std::string name1, std::string name2, ...)` should be called by all MPI processes.

`bool resume_mode()`: Returns true if the printers have resumed writing to the outputs of a previous scan. Generally, scanner plugins should take their cue on whether or not to resume a previous run from the printers.

At the heart of the printer system are the `printer` stream objects. These objects provide the necessary methods for printing values and associating them with a given point ID. The printer stream is manipulated using the following member functions:

`void reset(bool force)`: Deletes output that was already in the stream. By default, the main printer cannot be reset; to override this behaviour, set `force` to `true`.

`void print(`*value_type* `value, std::string name, int rank, unsigned long long int id)`: This function prints the actual output, sending a single datum of the given `value` and *value_type* to the printer. The output is identified as being the quantity `name`, and corresponding to the parameter combination uniquely identified by the point `id` and MPI `rank`.

Scanner-specific output files not associated with the GAMBIT printer system should typically be saved in the default scanner output path, which is accessed with `get_inifile_value<std::string>("default_output_path")`, and set to `scanner_plugins/`*plugin_name* by default.

[8] Note that `printer` is just a local ScannerBit `typedef` of the GAMBIT printer base class.

D.5: Scanner plugins

Scanner plugins receive access to an additional pair of utility functions and a class, for obtaining likelihood functors and scanner information:

`unsigned int get_dimension()`: Gets the dimension of the unit hypercube being explored.
`void* get_purpose(std::string purpose)`: Gets a pointer to a functor that is able to compute the quantity corresponding to `purpose`. In GAMBIT scans, purpose is conventionally `"LogLike"`, and the functor returned will be a direct conduit to the **likelihood container**.
`like_ptr`: A functor class used to contain the output of `get_purpose`, primarily designed to act as the local representation of the likelihood function within a plugin. A `like_ptr` can be called as if it were a function with signature `double (const std::vector<double>&)`. Typically, within a scanner plugin, the scanner passes a vector of unit hypercube parameter values to the `like_ptr`. This functor automatically performs any required prior transformation, computes the quantities corresponding to its `purpose`, and sends the corresponding quantities and hypercube parameters to the printer. The `like_ptr` member function `disable_external_shutdown()` can also be used from the plugin constructor to tell the objective function not to carry out its own shutdown procedure, but to simply set an internal `quit` flag (referred to in Ref. [29]) and rely on the scanner to terminate the scan itself.

D.5.1: Scanner plugin example

Here we give a simple example of a scanner plugin declaration, which closely follows one contained in the ScannerBit source code (`ScannerBit/src/scanners/random.cpp`). The example declares a scanner plugin named random, version 1.0.0-example. This scanner enters `number` random points in the functor corresponding to the purpose specified by the `like` YAML file option.

```
#include "scanner_plugin.hpp"

scanner_plugin(random, version(1, 0, 0, example))
{
  reqd_inifile_entries("number");

  like_ptr loglike;
  int num, dim;

  plugin_constructor
  {
    std::string purpose =
     get_inifile_value<std::string>("like")
    loglike = get_purpose(purpose);
    num = get_inifile_value<int>("number");
    dim = get_dimension();
  }

  int plugin_main(void)
  {
    std::vector<double> a(dim);
    for (int j = 0; j < num; j++)
    {
      for (int i = 0; i < dim; i++)
      {
        a[i] = Gambit::Random::draw();
      }
      loglike(a);
    }
    return 0;
  }

  plugin_deconstructor
  {
    std::cout << "no more plugin" << std::endl;
  }
}
```

The actual scanner code is declared within the `plugin_main` function, and randomly draws a parameter point from the hypergrid via the line

```
a[i] = Gambit::Random::draw();
```

When the plugin is loaded, the `plugin_constructor` function is run, initialising the variables `loglike`, `num`, and `dim`. The likelihood calculations and printing are done by the line `loglike(a)`. When the plugin is unloaded, the `plugin_deconstructor` function runs, and indicates to `stdout` that the plugin has been unloaded. At the top of the plugin declaration, the `reqd_inifile_entries("number")` macro indicates that the inifile entry `number` is required in order to use this scanner (see Sect. 3.2).

D.6: Objective plugins

In addition to the general plugin functions described in Sects. D.2–D.4, objective functions are provided with utility functions that can be used to probe the parameters being scanned, set the hypercube dimension and print parameters:

`std::vector<std::string>& get_keys()`: Retrieve the names of all the parameters being scanned over.
`void set_dimension(unsigned int dim)`: For plugins that will be used as priors. Sets the hypercube dimension that will be operated on by the prior to `dim`.
`void print_parameters(std::unordered_map<std::string, double> map)`: Prints the contents of a map from strings to double-precision floating-point variables. Typically used to print a set of parameters, where the map associates parameter names with their values.

D.6.1: Objective plugin example

Here we give a simple example of an objective plugin declaration contained in the ScannerBit source code (`ScannerBit/src/objectives/examples.cpp`). This example declares a scanner plugin EggBox, version 1.0.0. It returns a likelihood of the form:

$$\mathcal{P}(x, y) = \left[2 + \cos\left(\tfrac{\pi}{2}x\right)\cos\left(\tfrac{\pi}{2}y\right)\right]^5. \quad (2)$$

```
#include "objective_plugin.hpp"

objective_plugin(EggBox, version(1, 0, 0))
{
  std::pair <double, double> length;
  unsigned int dim;

  plugin_constructor
  {
    dim = get_keys().size();

    if (dim != 2)
    {
      scan_err << "EggBox: Need two parameters."
               << scan_end;
    }
    length = get_inifile_value<std::pair<double,
     double> > ("length", std::pair<double,
     double>(10.0, 10.0));
  }

  double plugin_main(std::unordered_map
   <std::string,double> &map)
  {
    print_parameters(map);

    double params[2];
    params[0] = map[get_keys()[0]]*length.first;
    params[1] = map[get_keys()[1]]*length.second;

    return 5.0*std::log(2.0 +
     std::cos(params[0]*M_PI_2) *
     std::cos(params[1]*M_PI_2));
  }
}
```

In the `plugin_constructor`, the hypercube dimension is obtained by testing how many parameters are returned from the `get_keys()` function. If the hypercube dimension does not match expectations, a runtime error is thrown with the `scan_err` and `scan_end` macros. The constructor initialises the scale length for each of the hypercube dimensions with the values assigned to the `length` key in the input YAML file. If no values are specified, both lengths default to 10. The `plugin_main` function does the actual likelihood calculation, as it is the function run by the scanner for every parameter combination. For each likelihood evaluation, the `plugin_main` receives an `unordered_map` with the parameter names and values, which it uses to compute the value of the likelihood. The contents of the map are printed with the command `print_parameters(map)`.

Appendix E: Scanner comparisons in a two-dimensional parameter space

The scanner comparisons presented in Sect. 11 are based on about 16 separate scans for each scanner in two, seven and fifteen dimensions. We also included results from 52 more scans to cover each dimensionality between two and fifteen. However, for clarity we only displayed two-dimensional profile likelihoods for the 15-dimensional scans (Figs. 7, 8, 9, 11, 12, 14 and 15). In this Section we present the equivalent plots to these for the two-dimensional scans. In some cases, where the optimal settings depends strongly on dimensionality, we have chosen different sampler settings in two than in fifteen dimensions, so as to allow a meaningful comparison.

E.1: MultiNest & Diver

The profile likelihoods for MultiNest and Diver are presented in Figs. 19 and 21 respectively. The marginalised posterior for MultiNest is given in Fig. 20. For both MultiNest and Diver, we present scans with the same settings as used for the 15-dimensional equivalent (Figs. 7, 8 and 9).

The quality of the profile likelihood is dramatically better in the two-dimensional scans than in the fifteen-dimensional equivalents. Although MultiNest did not sample the low-mass region at all in fifteen dimensions, it has been well sampled in two. The maximum likelihood point is located in the low-mass mode in all scans presented in Figs. 19 and 21. This is in good agreement with the analysis in Figs. 5 and 6, in which the maximum likelihood was easily achieved in two dimensions with less stringent scanner settings.

The marginalised posteriors in Fig. 20 show some qualitative differences to their 15-dimensional counterparts in Fig. 8. The primary difference is that the low-mass region shows in two dimensions, but not in fifteen. This is because in two dimensions, the low-mass region does not suffer from the same fine-tuning penalty (imposed by the integration over the nuisance parameters) as in fifteen dimensions. This penalty is due to the dependence of the exact location and shape of the low-mass region on the values of the 13 nuisance parameters included in the 15-dimensional scan. This reduces the ratio of the posterior mass of the low-mass mode to the pos-

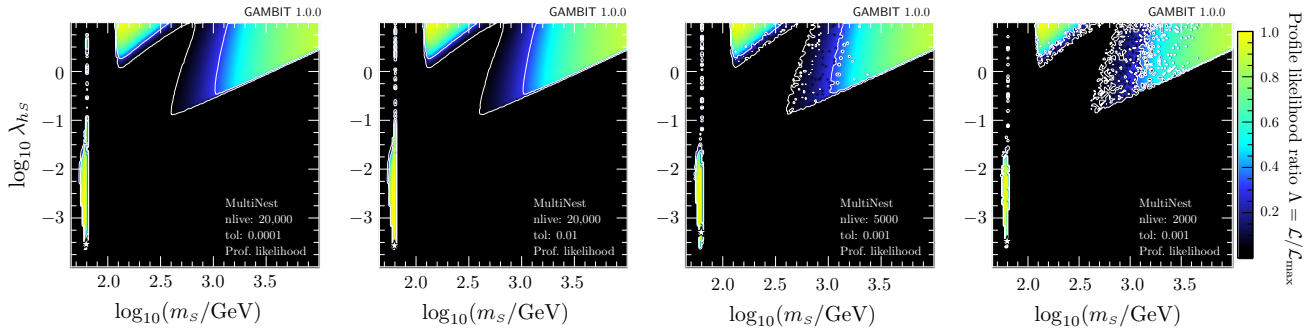


Fig. 19 Profile likelihood ratio maps from a 2-dimensional scan of the scalar singlet parameter space, using the MultiNest scanner with a selection of difference tolerances (`tol`) and numbers of live points (`nlive`). The maximum likelihood point is shown by a white star

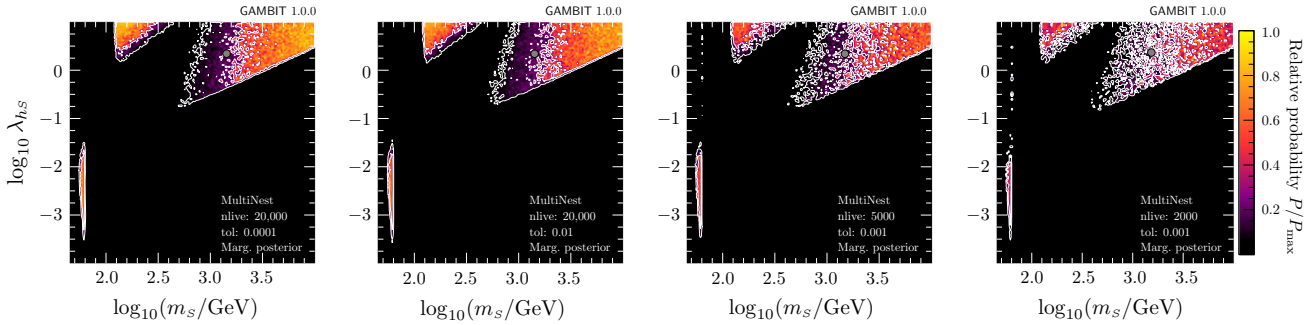


Fig. 20 Marginalised posterior probability density maps from a 2-dimensional scan of the scalar singlet parameter space, using the MultiNest scanner with a selection of difference tolerances (`tol`) and numbers of live points (`nlive`). Note that the colourbar strictly only applies to the rightmost panel, and that colours map to the same enclosed posterior mass on each plot, rather than to the same iso-posterior density level (i.e. the transition from red to purple is designed to occur at the edge of the 1σ credible region, and so on). The posterior mean is shown with a grey bullet point

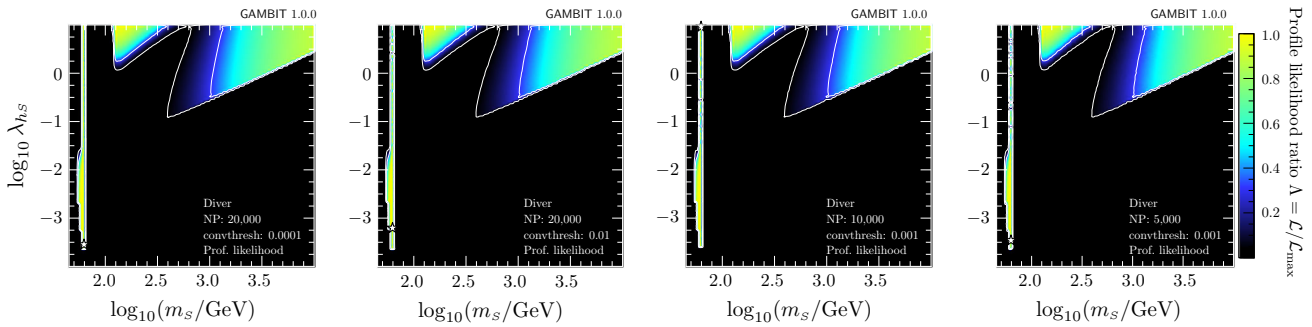


Fig. 21 Profile likelihood ratio maps from a 2-dimensional scan of the scalar singlet parameter space, using the Diver scanner with a selection of different convergence thresholds (`convthresh`) and population sizes (`NP`). The maximum likelihood point is shown by a white star

terior mass of the high-mass mode in the fifteen-dimensional scan.

E.2: T-Walk

The profile likelihoods and marginalised posteriors for two-dimensional T-Walk scans are presented in Figs. 22 and 23, respectively. We use different T-Walk settings compared to Figs. 11 and 12. This is primarily dictated by the dimensional dependence of the optimal number of chains, `chain_number`, as discussed in Sects. 11.3 and B.3. We find that values of `tol` ~ 0.1 causes very rapid convergence in two dimensions, even before any meaningful sampling can occur. This behaviour can be seen in the right-most plot of Fig. 22, where `tol` = 0.03. We therefore use different settings, more appropriate for the two-dimensional parameter space.

We find that T-Walk samples the profile likelihood very well in two dimensions when `tol` $\lesssim 0.01$. The number of chains appears to have less influence on the quality of the sampling, but dramatically increases the runtime. The

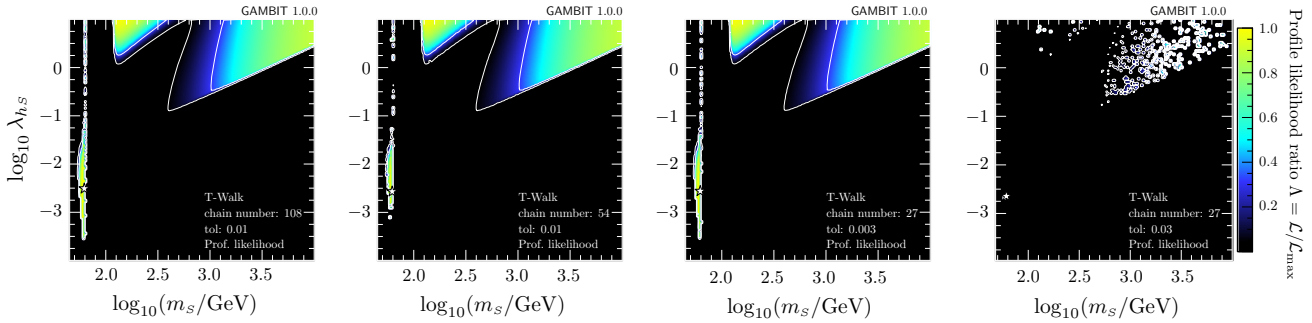


Fig. 22 Profile likelihood ratio maps from a 2-dimensional scan of the scalar singlet parameter space, using the T-Walk scanner with various numbers of chains and different tolerances. The maximum likelihood point is shown by a white star

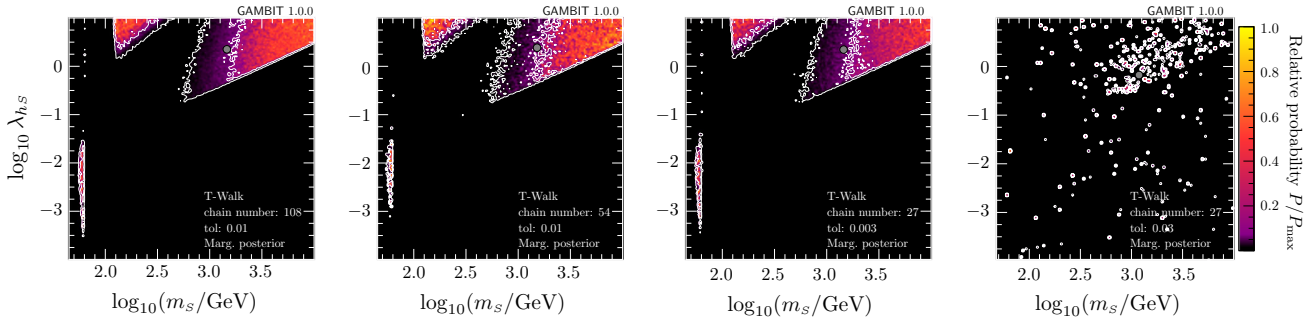


Fig. 23 Marginalised posterior probability density maps from a 2-dimensional scan of the scalar singlet parameter space, using the T-Walk scanner with various numbers of chains and different tolerances. Note that the colourbar strictly only applies to the rightmost panel, and that colours map to the same enclosed posterior mass on each plot, rather than to the same iso-posterior density level (i.e. the transition from red to purple is designed to occur at the edge of the 1σ credible region, and so on). The posterior mean is shown with a grey bullet point

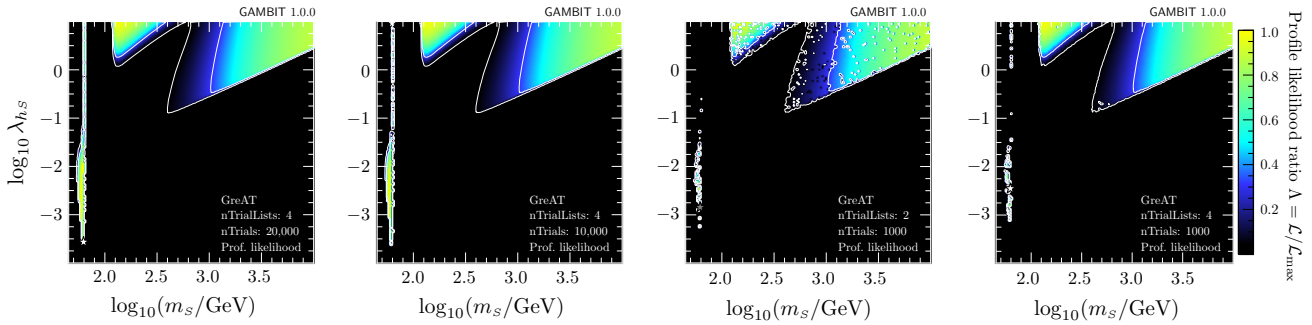


Fig. 24 Profile likelihood ratio maps from a 2-dimensional scan of the scalar singlet parameter space, using the GreAT sampler with various numbers of chains (`nTrialLists`) and chain lengths (`nTrials`). The maximum likelihood point is shown by a white star

scans of the two left-most plots in Fig. 22 took ∼4 h (`chain_number` = 54) and ∼18 h (`chain_number` = 108).

Although the sampling of the profile likelihood is much more complete in these two-dimensional scans than in the 15 dimensional case, there is no significant improvement in the marginalised posteriors (Fig. 23). However, we do see that the low-mass region appears within the two-sigma contours (as discussed in Sect. E.1).

E.3: GreAT

The profile likelihoods and marginalised posteriors for GreAT scans in a two-dimensional parameter space are presented in Figs. 24 and 25, respectively. The scanner settings in these plots are equivalent to those in Figs. 14 and 15, except for `nTrialLists`, which is set to $N_{\mathrm{dim}} = 2$ or $N_{\mathrm{dim}} + 2 = 4$.

The two left-most plots of Fig. 24 clearly show that these settings are excessive for sampling the profile likelihood in

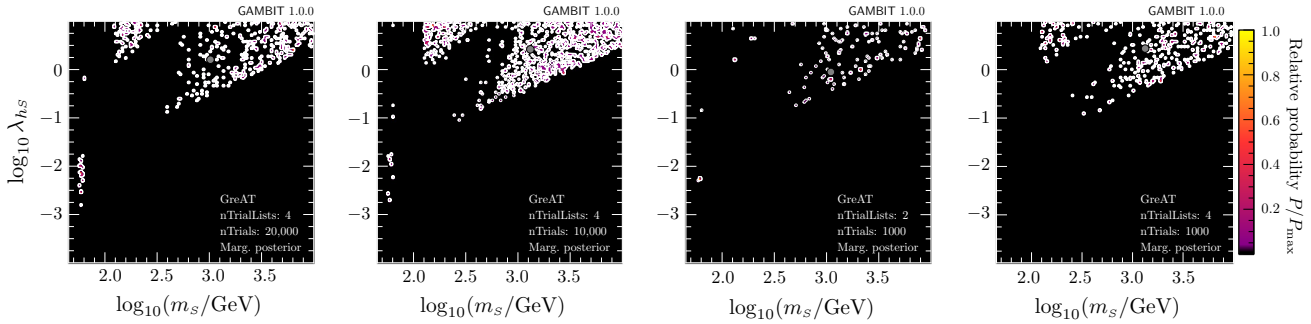


Fig. 25 Marginalised posterior ratio maps from a 2-dimensional scan of the scalar singlet parameter space, using the GreAT sampler with various numbers of chains (`nTrialLists`) and chain lengths (`nTrials`). Note that the colourbar strictly only applies to the rightmost panel, and that colours map to the same enclosed posterior mass on each plot, rather than to the same iso-posterior density level (i.e. the transition from red to purple is designed to occur at the edge of the 1σ credible region, and so on). The posterior mean is shown with a grey bullet point

two dimensions. Even though all panels in Fig. 24 exhibit well-sampled profile likelihoods, one can make an optimal choice when considering the computing time taken. From left to right, the scans took ∼5, 3 h, 8 and 17 min. Only in the quickest two scans does some degradation of the contours and sampling begin to appear. In contrast to the quality of the profile likelihood, we see in Fig. 25 that even with a long scan, in two dimensions the marginalised posterior is not well sampled by GreAT.

E.4: Summary

We have presented profile likelihoods and marginalised posteriors for scans of a two-dimensional parameter space, directly comparable to the 15-dimensional case presented in Sect. 11. These plots show that the inclusion of the additional 13 nuisance parameters does not significantly alter the joint profile likelihood of λ_{hS} and m_S. We find that sampling performance is significantly improved, demonstrating that although the additional 13 parameters are well constrained by unimodal likelihoods, their inclusion creates a significant challenge for the sampling algorithms.

Appendix F: YAML input file example

Below is an example YAML input file for ScannerBit_standalone that uses the custom prior defined in Appendix C, and the scanner and objective plugins declared in Appendices D.5.1 and D.6.1.

```
Parameters:
  EggBox:
    param_0:
      range: [0, 1]
    param_1:
      prior_type: dummy

Scanner:
  use_objectives: eggbox_like
  use_scanner: random_scanner

  objectives:
    eggbox_like:
      plugin: EggBox
      purpose: loglike
      length: [12, 12]

  scanners:
    random_scanner:
      plugin: random
      point_number: 2000
      like: loglike

Printer:
  printer: ascii
  options:
    output_file: "results.txt"

KeyValues:
  likelihood:
    model_invalid_for_lnlike_below: -1e5
```

Here, the model chosen for scanning is actually given as `EggBox`, which is the name of an objective plugin. Although we have not discussed such usage earlier in this paper, an objective plugin can in fact even be listed as a model when it provides the likelihood that is to be scanned, as is done here. This can be useful for avoiding any need to explicitly define a new model in GAMBIT format when all one is interested in is computing some external function provided by an objective plugin. In this case, the names given to parameters in the YAML file are entirely arbitrary. Here, the parameter `param_0` is defined to have a flat prior in the range `[0,1]`, and parameter `param_1` is defined to use the custom prior `dummy`. Next, the objective and scanner plugins are defined in the `Scanner` section. The `eggbox_like` objective is selected with the `use_objectives` directive, and the `random_scanner` scanner is selected as the chosen

scanner via the `use_scanner` directive. The `eggbox_like` objective is defined to use the `EggBox` plugin, with purpose set to `loglike`, and the option `length` set to `[0, 1]`. The `random_scanner` scanner is set to use the `random` plugin, with functions assigned the purpose `loglike` used to make up the likelihood function that it will call. The `point_number` option is set to ensure that `2000` samples are taken.

Appendix G: Glossary

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

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

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

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

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

capability A name describing the actual quantity that is calculated by a module or backend function. This is one possible place for units to be noted; the other is in the documented description of the capability (see Sect. 10.7 of Ref. [29]).

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

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

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

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.

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.

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.

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

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

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

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 of Ref. [29])

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.

scanner plugin An interface in ScannerBit to an external code for parameter sampling, i.e. a scanner.

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**.

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

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

References

1. C.F. Berger, J.S. Gainer, J.A.L. Hewett, T.G. Rizzo, Supersymmetry without prejudice. JHEP **2**, 23 (2009). arXiv:0812.0980
2. ATLAS Collaboration: 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. N. Christensen, R. Meyer, L. Knox, B. Luey, Bayesian methods for cosmological parameter estimation from cosmic microwave background measurements. Class. Quantum Gravity **18**, 2677–2688 (2001). arXiv:astro-ph/0103134

4. J. Dunkley, M. Bucher, P.G. Ferreira, K. Moodley, C. Skordis, Fast and reliable Markov chain Monte Carlo technique for cosmological parameter estimation. MNRAS **356**, 925–936 (2005). arXiv:astro-ph/0405462
5. 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
6. A. Lewis, S. Bridle, CosmoMC++, unpublished note (2006). http://cosmologist.info/notes/CosmoMC.pdf
7. 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
8. B.C. Allanach, C.G. Lester, Multidimensional mSUGRA likelihood maps. Phys. Rev. D **73**, 015013 (2006). arXiv:hep-ph/0507283
9. P. Bechtle, K. Desch, P. Wienemann, Fittino, a program for determining MSSM parameters from collider observables using an iterative method. Comp. Phys. Commun. **174**, 47–70 (2006). arXiv:hep-ph/0412012
10. R. Ruiz de Austri, R. Trotta, L. Roszkowski, A Markov chain Monte Carlo analysis of CMSSM. JHEP **5**, 2 (2006). arXiv:hep-ph/0602028
11. O. Buchmueller, R. Cavanaugh et al., Predictions for supersymmetric particle masses using indirect experimental and cosmological constraints. JHEP **9**, 117 (2008). arXiv:0808.4128
12. J. Skilling, Nested Sampling, in American Institute of Physics Conference Series, vol. 735, ed. by R. Fischer, R. Preuss, U.V. Toussaint, pp. 395–405 (2004)
13. 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
14. 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
15. Planck Collaboration, P.A.R. Ade, et. al., Planck 2015 results. XIII. Cosmological parameters. A&A **594**, A13 (2016). arXiv:1502.01589
16. 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
17. 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
18. 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
19. GAMBIT Collider Workgroup: C. Balázs, A. Buckley, et. al., ColliderBit: a GAMBIT module for the calculation of high-energy collider observables and likelihoods. Eur. Phys. J. C (in press) (2017). arXiv:1705.07919
20. 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
21. M. Ghulam, A. Faisal, M. Bilal, Optimization of neutrino oscillation parameters using differential evolution. Commun. Theor. Phys. **59**, 324–330 (2013). arXiv:1109.2431
22. 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
23. 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
24. 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
25. 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
26. P. Bechtle, J.E. Camargo-Molina et al., Killing the cMSSM softly. Eur. Phys. J. C **76**, 96 (2016). arXiv:1508.05951
27. A. Putze, L. Derome, The Grenoble Analysis Toolkit (GreAT)—a statistical analysis framework. Phys. Dark Univ. **5**, 29–34 (2014)
28. E .E .O. Ishida, S .D .P. Vitenti et al., COSMOABC: likelihood-free inference via population Monte Carlo approximate Bayesian computation. Astron. Comput. **13**, 1–11 (2015). arXiv:1504.06129
29. GAMBIT Collaboration: P. Athron, C. Balazs, et. al., GAMBIT: the global and modular beyond-the-standard-model inference tool. arXiv:1705.07908
30. GAMBIT Dark Matter Workgroup: T. Bringmann, J. Conrad, et. al., DarkBit: a GAMBIT module for computing dark matter observables and likelihoods. arXiv:1705.07920
31. 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
32. GAMBIT Flavour Workgroup: F. U. Bernlochner, M. Chrzaszcz, et. al., FlavBit: a GAMBIT module for computing flavour observables and likelihoods. arXiv:1705.07933
33. GAMBIT Collaboration: P. Athron, C. Balázs, et. al., Global fits of GUT-scale SUSY models with GAMBIT. arXiv:1705.07935
34. GAMBIT Collaboration: P. Athron, C. Balázs, et. al., A global fit of the MSSM with GAMBIT. arXiv:1705.07917
35. GAMBIT Collaboration: P. Athron, C. Balázs, et. al., Status of the scalar singlet dark matter model. arXiv:1705.07931
36. P. Scott, Pippi—painless parsing, post-processing and plotting of posterior and likelihood samples. Eur. Phys. J. Plus **127**, 138 (2012). arXiv:1206.2245
37. N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, E. Teller, Equation of state calculations by fast computing machines. J. Chem. Phys. **21**, 1087–1092 (1953)
38. D. MacKay, *Information theory, inference, and learning algorithms* (Cambridge University Press, Cambridge, 2003). (ISBN:0521642981)
39. R.M. Neal, Probabilistic inference using Markov Chain Monte Carlo methods. Technical Report CRG-TR-93-1 (1993)
40. W.K. Hastings, Monte Carlo sampling methods using Markov chains and their applications. Biometrika **57**, 97–109 (1970)
41. J.A. Christen, J. Weare, A general purpose sampling algorithm for continuous distributions (the t-walk). Bayesian Anal. **5**, 263 (2010)
42. J. Goodman, J. Weare, Ensemble samplers with affine invariance. Commun. App. Math. Comput. Sci. **5**, 65 (2010)
43. A. Gelman, D.B. Rubin, Inference from iterative simulation using multiple sequences. Stat. Sci. **7**, 457–472 (1992)
44. R. Storn, K. Price, Differential evolution: a simple and efficient heuristic for global optimization over continuous spaces. J. Glob. Optim. **11**, 341–359 (1997)
45. K. Price, R.M. Storn, J.A. Lampinen, *Differential evolution: a practical approach to global optimization* (Springer, Berlin, 2005)
46. S. Das, P. Suganthan, Differential evolution: a survey of the state-of-the-art. Evolut. Comput. IEEE Trans. **15**, 4–31 (2011)
47. K. Price, Differential evolution, in *Handbook of Optimization. Intelligent Systems Reference Library, vol. 38*, ed. by I. Zelinka, V. Snášel, A. Abraham (Springer, Berlin, 2013), pp. 187–214
48. K. Price, R.M. Storn, J.A. Lampinen, The differential evolution algorithm, in *Differential Evolution: A Practical Approach to Global Optimization*, Natural Computing Series (Springer, Berlin, 2005), pp. 37–134
49. D. Zaharie, A comparative analysis of crossover variants in differential evolution. Proc. IMCSIT **2007**, 171–181 (2007)

50. D. Zaharie, Statistical properties of differential evolution and related random search algorithms, in *COMPSTAT 2008*, ed. by P. Brito (Physica-Verlag, Heidelberg, 2008), pp. 473–485
51. E. Mezura-Montes, J. Velázquez-Reyes, C.A. Coello Coello, A comparative study of differential evolution variants for global optimization, in *Proceedings of the 8th annual conference on Genetic and evolutionary computation, GECCO '06* (ACM, New York, 2006), pp. 485–492
52. D. Zaharie, Influence of crossover on the behavior of differential evolution algorithms. Appl. Soft Comput. **9**, 1126–1138 (2009)
53. J. Brest, S. Greiner, B. Boskovic, M. Mernik, V. Zumer, Self-adapting control parameters in differential evolution: a comparative study on numerical benchmark problems. Evolut. Comput. IEEE Trans. **10**, 646–657 (2006)
54. F. Neri, V. Tirronen, Recent advances in differential evolution: a survey and experimental analysis. Artif. Intell. Rev. **33**, 61–106 (2010)
55. 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
56. A. Beniwal, F. Rajec et al., Combined analysis of effective Higgs portal dark matter models. Phys. Rev. D **93**, 115016 (2016). arXiv:1512.06458
57. J.M. Cline, K. Kainulainen, P. Scott, C. Weniger, Update on scalar singlet dark matter. Phys. Rev. D **88**, 055025 (2013). arXiv:1306.4710
58. 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
59. Y. Mambrini, Higgs searches and singlet scalar dark matter: combined constraints from XENON 100 and the LHC. Phys. Rev. D **84**, 115017 (2011). arXiv:1108.0671
60. C.P. Burgess, M. Pospelov, T. ter Veldhuis, The minimal model of nonbaryonic dark matter: a singlet scalar. Nucl. Phys. B **619**, 709–728 (2001). arXiv:hep-ph/0011335
61. J. McDonald, Gauge singlet scalars as cold dark matter. Phys. Rev. D **50**, 3637–3649 (1994). arXiv:hep-ph/0702143
62. V. Silveira, A. Zee, Scalar phantoms. Phys. Lett. B **161**, 136–140 (1985)
63. Particle Data Group: K. A. Olive et al, Review of Particle Physics. Chin. Phys. C **38**, 090001 (2014)
64. Particle Data Group: K.A. Olive et. al., Review of Particle Physics, update to Ref. [62] (2015). http://pdg.lbl.gov/2015/tables/rpp2015-sum-gauge-higgs-bosons.pdf
65. M. Cacciari, G.P. Salam, G. Soyez, FastJet user manual. Eur. Phys. J. C **72**, 1896 (2012). arXiv:1111.6097
66. P. Athron, J.-H. Park, D. Stöckinger, A. Voigt, FlexibleSUSY—a spectrum generator for supersymmetric models. Comput. Phys. Commun. **190**, 139–172 (2015). arXiv:1406.2319
67. B.C. Allanach, SOFTSUSY: a program for calculating supersymmetric spectra. Comput. Phys. Commun. **143**, 305–331 (2002). arXiv:hep-ph/0104145